KAPI

Programmer's Guide

January 2001

Document Number 748282.002

World Wide Web: http://developer.intel.com

Version	Version History	Date
-002	Updated to use the Itanium™ trademark	January 2001

Contents

Introduction 1

A knobsfile is a data file that describes various microarchitecture details. As the complexity of Iintel® Itanium™ microarchitectures increases, it is important for Itanium architecture tools (compilers, simulators, and performance tools) to operate efficiently regardless of the target microarchitecture.

The information in this document deals with three layers:

KAPI layer — This first layer is the parser. It reads the knobsfile and translates the symbols and syntax to C structures.

KAPI Itanium architecture layer — The second layer interprets the C structures as microarchitectural information to obtain knobsfile values.

KMAPI layer — The third layer is the machine-state information layer. It interfaces to the rules and algorithms.

This document describes KAPI, which is an API that allows other tools to read and interpret the knobsfile format. Use of a single API

- enables access to information about Itanium microarchitecture details for abstraction
- permits transition from one generation of processor to another
- provides a standardized definition of basic microarchitectural modelling terminology and conventions
- improves the modelling consistency between Itanium architecture tools
- eliminates the need for developing different configuration files and models for different Itanium architecture tools

This document defines specific models and conventions used to model Itanium microarchitectures.

NOTE. *The information represented by KAPI and its associated knobsfiles is approximate and may not be able to completely represent a given Itanium processor's exact behavior, latency, or resources given the complexity of microprocessor designs. Tools that require 100% accuracy for correct behavior should not use KAPI or its knobsfiles.*

About This Manual

This document contains the following chapters:

- Chapter 1, "Introduction", introduces this document and KAPI, and lists the manuals referred to in this document.
- Chapter 2, "KAPI Format and Semantics", details the first KAPI layer.
- Chapter 3, "Accessing Generic Knobsfile Data", details the functions of the first KAPI layer.
- Chapter 4, "Abstract Model for Intel® Itanium™ Microarchitectures", provides KAPI definitions and an abstract model.
- Chapter 5, "Accessing Intel® Itanium™ Architecture Information", describes KAPI conventions and header files for the first and second layers.
- Chapter 6, "Defining Intel® Itanium™ Microarchitecture", defines the variables and attributes that define Itanium microarchitectures.
- Chapter 7, "KAPI Functions", describes the KAPI functions, parameters, and return values.
- Chapter 8, "KMAPI: For Deriving Machine-State Information", details the KMAPI functions for the third KAPI layer.

Related Publications

Refer to the following documentation, available at http://developer.intel.com, for more information:

Intel® Itanium™ Architecture Software Developer's Manual

- *Volume 1: Application Architecture*
- *Volume 2: System Architecture*
- *Volume 3: Instruction Set Reference*
- *Volume 4: Itanium Processor Programmer's Guide*

Itanium™ Processor Microarchitecture Reference for Software Optimization

Notational Conventions

`this type style`	Indicates code. Object names may appear in upper and lowercase.
`this type style`	Indicates a parameter.
`[items]`	Indicates that the items enclosed in brackets are optional.
`THIS TYPE STYLE`	Indicates enumeration.
this type style	Indicates a hypertext link.

Files

KAPI needs the following header files:

- `sched_ia64.h`
- `sched_enums.h`

KAPI needs the following library files:

- `libsched.a` (or `.lib` for Windows NT*) for static linking
- `libsched.so` (or `sched.lib` and `sched.dll` for Windows NT) for dynamic linking

KAPI Format and Semantics

2

The knobsfile format is a simple programming language for defining values of scalar variables, arrays, and high dimensional data (called attributes). As with normal sequential languages, variable assignments override earlier ones and identifiers must be defined before they can be used. Sometimes definition, declaration, and assignment can be done with a single statement.

To read a knobsfile, the `KAPI_Initialize` function is called. This function reads, parses, and computes the values of the defined variables, types, and attributes. If any errors are detected, the initialize function fails.

Other KAPI functions are provided for querying these values.

Simple Assignments

Simple scalar variables do not need to be declared and can be assigned with a assignment statement.

This is an integer variable declaration/assignment:

```
VariableName := 5;
```

The left-hand side (lhs) type is determined by the right-hand side (rhs). Following is another statement:

```
VariableName := “test“;
```

This is an implicit type string. The string data type starts with a double quote and extends to the next double quote. Strings cannot wrap beyond one line.

There are three predefined variable types: `int`, `real`, and `string`. You can use simple arithmetic expressions involving plus (+), although only with integer values; e.g.:

```
VariableName := 5 + 3; Valid
```

```
VariableName := 5.2 + 3; Invalid
VariableName := "a" + "b"; Invalid
```

You can use variables that were assigned an integer value on the right hand side of an assignment, as in:

```
NewVariableName := VariableName + 1;   OK if VariableName is int
```

CAUTION. *When using a variable on the right hand side of an assignment, the used variable should not be reassigned a different value later. If it is assigned a different value later, the variable on the left hand side is NOT updated, and a warning message is issued.*

Type Definitions and Non-Scalar Assignments

The knobsfile allows enumerated types to be declared and used, both for naming values and for indexing arrays. This is a declaration of an enumerated type with three values:

```
type colors_t = enum (colRed, colBlue, colGreen);
```

This declaration has two effects. First, it declares a new type called `colors_t` that has three elements. Second, it defines three identifiers (`colRed`, `colBlue`, and `colGreen`) that can be used in the knobsfile after this declaration. These names cannot be reused in other type statements; i.e., an enumeration constant belongs to only one type.

Once an enumerated type has been declared, assign a value to the variables:

```
MyColor := colRed;
```

Note that the type declaration is optional in this assignment statement. Enumerated types can also declare arrays:

```
Lumen : array[ color_t ] of real;
```

The above statement declares an array of three elements, each of which is real. Values are assigned as follows:

```
Lumen[ colRed ] := 1.3;
```

It is also possible to assign all elements to a default value:

```
Lumen[ * ] := 0.0;
```

Arrays can also be indexed by strings (associative arrays) and are declared as follows:

```
ALumen : array[ string ] of real;
```

Such arrays can then be set:

```
ALumen[ "sldkf" ] := 34.2;
```

To declare bitmask variables or arrays of bitmask variables:

```
BB : bitmask( color_t );
BBarray : array[ X_t ] of bitmask( color_t );
```

where `X_t` is some other enumerated type name that indexes the array. Bitmasks are formed by listing a subset of enumerated values of the specified base type; in this case, `color_t`:

```
BB := bitmask( colRed, colBlue ).
```

Type and Assignment Statement Level Limiting

This section discusses limits on variables and types.

Variable Limiting

The knobsfile can limit individual assignment statement values on a per statement basis. For example:

```
SomeVariable := 5
limit <"sphinx", "emerald"> ( 1,2,5,6 )
limit <"gluon" > ( 1 .. 8 )
limit <"foobar" > noredefine;
```

This assignment means that the default value is 5. For Sphinx, it can be later changed, but only to the values listed in the limit statement. Similarly, for Gluon, the value can be changed, but must be in the range from 1 to 8. Finally, the tool "foobar" is limited so that the value can only be 5. A later assignment statement cannot change "foobar".

It is possible to limit variable values of all tool names, using empty brackets:

```
SomeVariable := 5
limit <> ( 1, 4, 5 );
```

There are currently several limitations on the use of the `limit` clause:

- Variable limitation are available only for integers and ranges.
- Bitmask types cannot be value-limited.
- If an enumerated type is used in a limit statement, then the enumerated type cannot be redefined later.
- A limit clause cannot be used on a variable whose value is derived from an arithmetic expression.

Type Limiting

The `limit` statement can be used on type definitions, but only to indicate that the type cannot be redefined. No value ranges are allowed.

```
TYPE color_t = enum ( colRed, colBlue, colGreen )
limit <> noredefine;
```

Expect Statement

The `expect` statement provides rudimentary error checking. The `expect` statement lists the names of variables and types that must be defined prior to the end of the baseline and delta knobsfiles. The `expect` statement format is:

```
expect variable variable_name;
expect type type_name;
expect enum enumeration_const;
expect attribute attribute_name;
```

If any of the names listed in an `expect` statement are not defined in the knobsfiles, or are defined as a different type, an error message is printed, and initialization fails.

Append Statement

The `append` statement allows extending types.

The format is:

```
append type_name = enum( id1, id2 … )
```

For example:

```
append colors_t = enum ( colYellow, colMagenta )
```

This statement adds the identifiers to the type. The same rules applying to identifiers defined with the `type` statement apply to identifiers defined with the `append` statement.

You cannot append identifiers to a limited type (noredefine)

Implicit Statement

This is the `implicit none` statement:

```
implicit none;
```

This statement requires that all variables and attributes referenced after the `implicit` statement must be explicitly declared prior to their use. This forces you to set-up the knobsfile with variable declarations at the beginning, and avoids the problem of an undetected variable reference misspelling.

The default behaviour is to implicitly declare variables and attributes.

Attributes

Attributes enable entering highly irregular data. Primarily, a sequence of attribute values are listed in the knobsfile as strings and then accessed through the attribute functions that return their string values. The caller then interprets the contents of these attributes.

For example, in the previous knobsfiles versions, the instruction set features and bypass matrix were implemented using this technique since it represents high-dimension data that is difficult to express with scalar variables:

```
INTRACLUSTER += “fuIALU/primary:fuLD/primary/cportM1=1“;
INTRACLUSTER += “fuIALU/primary:fuFLD=1“;
INTRACLUSTER += “fuIALU/primary:fuFLDP=1“;
```

These values can then be retrieved in a C program by writing:

```
for (i=0;
    i<KAPI_count4attribute( pConfig,“INTRACLUSTER“ );
```

```
    i++ ) {
        string = KAPI_attribute4index( pConfig, "BYPASS", i );
    /* process string here */
}
```

The returned order of the attributes corresponds to the order listed in the knobsfile. Duplicates are possible.

CAUTION. *It is the caller's responsibility to interpret the contents of attribute variables and to check their contents for errors.*

Attributes can be declared prior to use in the same manner as variables are declared.

```
attribute_name : attribute.
```

Usage Issue Summary

These are the usage guidelines:

- The value of a variable or type when the knobsfile data is accessed through the API is its last assigned value.
- The type name of a variable may not be reassigned once the variable is declared or used.
- Types can be redefined. Any variables or types that are dependent upon that type may also need to be redefined. All arrays that are indexed by that type, change as follows:
  - — If an enumeration constant is removed from the index type by a redefinition, then the values of all array elements at that index are undefined and unavailable for access after the redefinition.
  - — If an enumeration constant is added to the index type of an array, then the values of all array elements at that index are considered to be uninitialized (unless a '*' initializer was used on the array previously).

  - If an enumeration constant was in both the original and in the newly redefined index type of an array, then such elements retain the same values as they had previously (even if the ordering of the enumeration constants was changed).
- When redefining a type all variables of that type behave as follows:
  - Any variable that was previously assigned the value of an enumeration constant that was removed by a type redefinition, becomes undefined (including array elements).
  - Any variable that was previously assigned the value of an enumeration constant that is in both the original and redefinition is retained.
- When redefining a type, all variables that are bitmasks based on that type behave as follows:
  - Any bit that corresponds to an enumerated constant that was removed is turned off.
  - All bits that correspond to retained enumerated values are retained.
  - Bits that correspond to new enumerated values are zero by default.
  - If the order of enumerated constants in a declaration is changed, then so does the implied position of bits in the bitvector. Positions can always be queried using the `KAPI_EnumIndex` function.
- Limited variables can be reassigned, but only to one of the specified values.
- In the current implementation, only arithmetic expressions containing plus are allowed, and only for integer type variables.
- In the current implementation, only integer type variables and integer values can be used on the right hand side in arithmetic expressions.

Accessing Generic Knobsfile Data

3

This chapter describes functions that enable reading knobsfiles and querying values of variables, types, and attributes. These functions are part of the first KAPI layer, and have no specific relationship to Intel® Itanium™ architecture. Later chapters describe the Itanium architecure-specific extension layer.

The functions in this chapter are grouped as follows:

- initialize and close
- generic data query
- complex data structures

All constants, variables, and types mentioned in this chapter and in Chapter 8, "KMAPI: For Deriving Machine-State Information" are defined in the KAPI header files.

Some fixed-length bitvectors are used for encoding various quantifiers. For example, the following type represents bitvectors, where *X* can be `32`, `64`, or `128`.

```
typedef int bvX_t;
```

Arbitrary-length bitvector pointer types can also be used. The following type is used for some functions that do not fit well in fixed-length sizes.

```
typedef void *bv_t;
```

See the `kapi_bv.h` header file for more information on the use of these bitvector functions.

All KAPI functions allocate copies of strings or values passed as return values. The caller is responsible for deallocating strings and values returned from KAPI functions.

Initialize & Close Functions

Use the following functions to start and stop KAPI.

KAPI_Initialize( )

Initializes KAPI

```
void *KAPI_Initialize( FILE *fpDelta, FILE *fpBaseline,
char *toolname)
```

Parameters

fpDelta	Knobsfile containing additional knobs.
fpBaseline	A knobsfile containing Itanium architecture knobs definitions.
toolname	A string indicating a specific tool, used where specific knobs are limited to specific tools.

Description

The KAPI_Initialize() function must be called once before any other KAPI functions in order to read configuration files, initialize state, and compute global values. The value returned by this function must be passed to all other interface functions. The *toolname* argument specifies how to interpret information in the knobsfile (some tools can ignore certain entries). If the value of *toolname* is not understood by KAPI or is NULL, a default tool name is used.

The *fpBaseline* parameter points to a file that contains an entire valid knobsfile that is parsed and read by the initialization function. The *fpDelta* file points to a file that contains set of overrides to the baseline settings. Either the delta or baseline arguments may be NULL, but not both.

The version number of both the delta and baseline must match, or KAPI returns a `NULL` pointer. The baseline file may define a set of variables whose values cannot be overwritten or definitions that limit the allowed range of values of some variables. If the delta file tries to override or violate any of these settings, initialization fails and error messages are returned.

Any errors in either input file cause a `NULL` pointer to be returned.

KAPI_ia64_Initialize( )

Initializes Itanium architecture data

```
void *KAPI_ia64_Initialize(void *pConfig)
```

Parameters

pConfig A pointer to the variable initialized by `KAPI_initialize()`.

Description

The `KAPI_ia64_Initialize()` function must be called once before working with Itanium architecture-specific interface functions. It builds the necessary data.

KAPI_Finalize( )

Releases resources

```
void KAPI_Finalize( void *pConfig )
```

Parameters

pConfig A pointer to the variable initialized by `KAPI_initialize()`.

Description

The `KAPI_Finalize()` function releases all resources allocated internally by KAPI for the specified configuration. Attempting to access any information from the configuration after this function has been called may result in incorrect answers or program failure.

Generic Data Query Functions

As the knobsfile is a constantly evolving interface and new microarchitectural features are likely to come and go, it is important to be able to query data in a generic format that allows new attributes to be added to the knobsfile without updating the functions.

This section provides functions that retrieve attributes by name in a variety of formats. Notice that many of these functions require integer indices. Determine the value of these indices by converting enumerated type values to integers using the functions in this section.

The following functions convert type and enumerated constant names to integers and integers to enumerated constant names based on definitions in the knobsfile. Use these functions to translate enumerated constant names to integers rather than assuming the knobsfile is ordered a specific way.

KAPI_EnumIndex( )

Gets KAPI index for enum string

```
extern int KAPI_EnumIndex( void *pConfig, char *pchType, char
*pchEnumName )
```

Parameters

pConfig	A pointer to the variable initialized by `KAPI_initialize()`.
pchType	A string representing the enumeration type.
pchEnumName	A string representing the enumeration name.

Description

Given a type name and an enumeration string constant, the `KAPI_EnumIndex()` function returns its integer representation. This function returns `-1` if no such type exists or no such enumeration constant is defined as part of it.

Example

`it_t` is defined in the knobsfile as follows:

```
TYPE it_t = enum ( itA, itM, itB, itBl, itI, itF, itL )
```

The function `KAPI_EnumIndex(pConfig, "it_t", "itA" )` returns `0` to indicate that `itA` is the first item in `it_t`.

KAPI_ArrayIndex( )

Gets KAPI index for array string index

```
extern int KAPI_ArrayIndex( void *pConfig, char *pchArray,
char *pchIndex)
```

Parameters

pConfig	A pointer to the variable initialized by `KAPI_initialize()`.
pchArray	A string representing the array name.
pchIndex	A string representing the index to the associative array.

Description

Given an associative array name and the string index (not including double quotes), the `KAPI_ArrayIndex()` function returns an integer value that can be passed to the `KAPI_Get`*XXX*`Variable`*XXX* functions to retrieve array element values.

Example

The following code returns the index for `fuST` in the `fuLatency` array, which can then be used to retrieve core latency for `fuST`.

```
KAPI_ArrayIndex(pKapiInfo, "fuLatency", "fuST");
```

KAPI_EnumCardinality( )

Gets cardinality for an enum

```
extern int KAPI_EnumCardinality( void *pConfig, char *pchType )
```

Parameters

pConfig	A pointer to the variable initialized by `KAPI_initialize()`.
pchType	A string representing the enumeration type.

Description

For a given enumeration type, the `KAPI_EnumCardinality()` function returns its cardinality. It returns `-1` if no such type exists.

KAPI_EnumName( )

Gets string for enum index

```
extern char *KAPI_EnumName( void *pConfig, int enumconst,
char *pchType )
```

Parameters

pConfig	A pointer to the variable initialized by `KAPI_initialize()`.
enumconst	A number representing the enumeration index.
pchType	A string representing the enumeration type.

Description

Given an enumeration integer constant and a type name, the `KAPI_EnumName()` function returns the string representation of that constant from the knobsfile. It returns `NULL` if no such name exists.

Example

`it_t` is defined in the knobsfile as follows:

```
TYPE it_t = enum ( itA, itM, itB, itBl, itI, itF, itL )
```

The function `KAPI_EnumName(pConfig, 0, "it_t" )` returns `itA` to indicate that `itA` is the first item appearing in `it_t`.

KAPI_VariableCardinality( )

Gets number of elements for attribute

```
extern int KAPI_VariableCardinality(void *pConfig, char *pchAttribute)
```

Parameters

pConfig	A pointer to the variable initialized by `KAPI_initialize()`.
pchAttribute	A string identifying the required attribute.

Description

The `KAPI_VariableCardinality()` function returns the number of elements in the named array. It returns `1` if *pchAttribute* is scalar or an array with only one element. If no such *pchAttribute* exists, it sets the `KAPI_error_attribute` flag to `-1`.

KAPI_GetEnumVariableName( )

Gets enum name

```
extern char *KAPI_GetEnumVariableName( void *pConfig,
char *pchAttribute, int iIndex )
```

Parameters

pConfig	A pointer to the variable initialized by `KAPI_initialize()`.
pchAttribute	A string identifying the required enumeration.
iIndex	Denotes the required index if the attribute has more than one value. Otherwise, use 0.

Description

Given a variable declared as an enumerated type, the `KAPI_GetEnumVariableName()` function returns the corresponding enumerated value name for the specified index. If *pchAttribute* is an array, it returns the indicated element of that array (arrays are zero-based). If *pchAttribute* is scalar, *iIndex* must be `0`. If no such *pchAttribute* exists or its value is undefined in the knobsfile, the `KAPI_error_attribute` flag is set to `-1`.

KAPI_GetEnumVariable( )

Gets enum value

```
extern int KAPI_GetEnumVariable( void *pConfig, char *pchAttribute,
int iIndex )
```

Parameters

pConfig	A pointer to the variable initialized by KAPI_initialize().
pchAttribute	A string identifying the required enumeration.
iIndex	Denotes the required index if the attribute has more than one value. Otherwise, use 0.

Description

Given a specific variable name that corresponds to an enumerated value, the KAPI_GetEnumVariable() function returns the corresponding integer value. If *pchAttribute* is an array, it returns the indicated element of that array (arrays are zero-based).

Use this function with care. If the order of the enumerated type changes in the knobsfile, the returned values may change. It is recommended that applications use the KAPI_GetEnumVariableName function instead. If *pchAttribute* is scalar, *iIndex* must be 0. If no such *pchAttribute* exists or its value is undefined in the knobsfile, the KAPI_error_attribute flag is set to -1.

KAPI_GetIntegerVariable( )

Gets int from name or attribute

```
extern int KAPI_GetIntegerVariable( void *pConfig, char *pchAttribute,
int iIndex )
```

Parameters

pConfig	A pointer to the variable initialized by `KAPI_initialize()`.
pchAttribute	A string identifying the required attribute.
iIndex	Denotes the required index if the attribute has more than one value. Otherwise, use 0.

Description

Given a specific variable name that corresponds to an integer value, the `KAPI_GetIntegerVariable()` function returns the value. If no such *pchAttribute* exists, it sets the `KAPI_error_attribute` flag to `-1`. If *pchAttribute* is an array, it returns the indicated element of that array (arrays are zero-based). If *pchAttribute* is scalar, *iIndex* must be `0`.

KAPI_GetDoubleVariable( )

Gets fp from name or attribute

```
extern double KAPI_GetDoubleVariable( void *pConfig,
char *pchAttribute, int iIndex )
```

Parameters

pConfig	A pointer to the variable initialized by `KAPI_initialize()`.
pchAttribute	A string identifying the required attribute.
iIndex	Denotes the required index if the attribute has more than one value. Otherwise, use 0.

Description

The `KAPI_GetDoubleVariable()` function gets the floating-point value associated with the specified array name. If no such array exists, it sets the `KAPI_error_attribute` flag to `-1`. If *pchAttribute* is an array, the function returns the indicated element of that array (arrays are zero-based). If *pchAttribute* is scalar, *iIndex* must be `0`.

KAPI_GetStringVariable( )

Gets string from name or attribute

```
extern char *KAPI_GetStringVariable( void *pConfig,
char *pchAttribute, int iIndex )
```

Parameters

pConfig	A pointer to the variable initialized by `KAPI_initialize()`.
pchAttribute	A string identifying the required attribute.
iIndex	Denotes the required index if the attribute has more than one value. Otherwise, use 0.

Description

The `KAPI_GetStringVariable()` function gets the string value associated with the specified array name. If no such array exists, it sets the `KAPI_error_attribute` flag to `-1`. If *pchAttribute* is an array, it returns the indicated element of that array (arrays are zero-based). If *pchAttribute* is scalar, *iIndex* must be 0.

KAPI_GetBvVariable( )

Gets bv name or attribute

```
extern bv_t *KAPI_GetBvVariable( void *pConfig, char *pchAttribute,
int iIndex )
```

Parameters

pConfig	A pointer to the variable initialized by KAPI_initialize().
pchAttribute	A string identifying the required attribute.
iIndex	Denotes the required index if the attribute has more than one value. Otherwise, use 0.

Description

The KAPI_GetBvVariable() function returns a pointer to the value associated with the specified variable name. Space is internally allocated by the interface for the bv_t structure. The caller is responsible for deallocating it when finished with the appropriate bitvector deallocation function. If no such *pchAttribute* exists, it sets the KAPI_error_attribute flag to -1 and returns NULL. If *pchAttribute* is an array, it returns the indicated element of that array (arrays are zero-based). If *pchAttribute* is scalar, *iIndex* must be 0.

Complex Data Structure Functions

These functions are used to access data with structure too complex to represent as simple scalar values or one-dimensional arrays. As such, the knobsfile author can simply list a sequence of lines prefixed with an attribute name followed by a string. Such entries in the knobsfile are referred to as attributes rather than variables or types. The API makes no attempt to interpret the contents of attributes, but only ensures that the order in which attributes appear in the knobsfile is the same as the indexing order used to access them.

KAPI_count4attribute( )

Gets variable/attribute cardinality

```
extern int KAPI_count4attribute( void *pConfig, char *pchAttribute )
```

Parameters

pConfig	A pointer to the variable initialized by `KAPI_initialize()`.
pchAttribute	A string identifying the required attribute.

Description

The `KAPI_count4attribute()` function returns the number of definitions of the name that *pchAttribute* points to in the knobsfile(s).

KAPI_attribute4index( )

Gets string from name or attribute

```
extern char *KAPI_attribute4index( void *pConfig, char *pchAttribute,
int iIndex)
```

Parameters

pConfig	A pointer to the variable initialized by `KAPI_initialize()`.
pchAttribute	A string identifying the required attribute.
iIndex	Denotes the required index if the attribute has more than one value. Otherwise, use 0.

Description

Given an index with a value between `0` and `KAPI_count4attribute() - 1` and an attribute name, the `KAPI_attribute4index()` function returns the string value stored in that attribute entry.

Abstract Model for Intel® Itanium™ Microarchitectures

4

This chapter describes the abstract model that is assumed for Intel® Itanium™ architecture knobsfiles, and defines the central terms for the first and second layers. The model described is expressed in Chapter 5, "Accessing Intel® Itanium™ Architecture Information" as a set of KAPI usage conventions. The variables, types, and attributes with well-known names that represent and control specific microarchitectural attributes are described in Chapter 6, "Defining Intel® Itanium™ Microarchitecture".

Terms

The following terms are used extensively throughout this document.

Port, cport, and cutport indexes all indicate the entry point to a functional unit.

cport	cluster-dependent port index
port	machine wide port index
cutport	cluster and functional unit dependent port index
fu	functional unit class of instruction producing a result (the producer functional unit class) or consuming a result (the consumer fuctional unit class)
operand	source or destination operand
cluster	a collection of functional units
latency	the number of delay cycles until the result of an operation can be used
ut	unit type

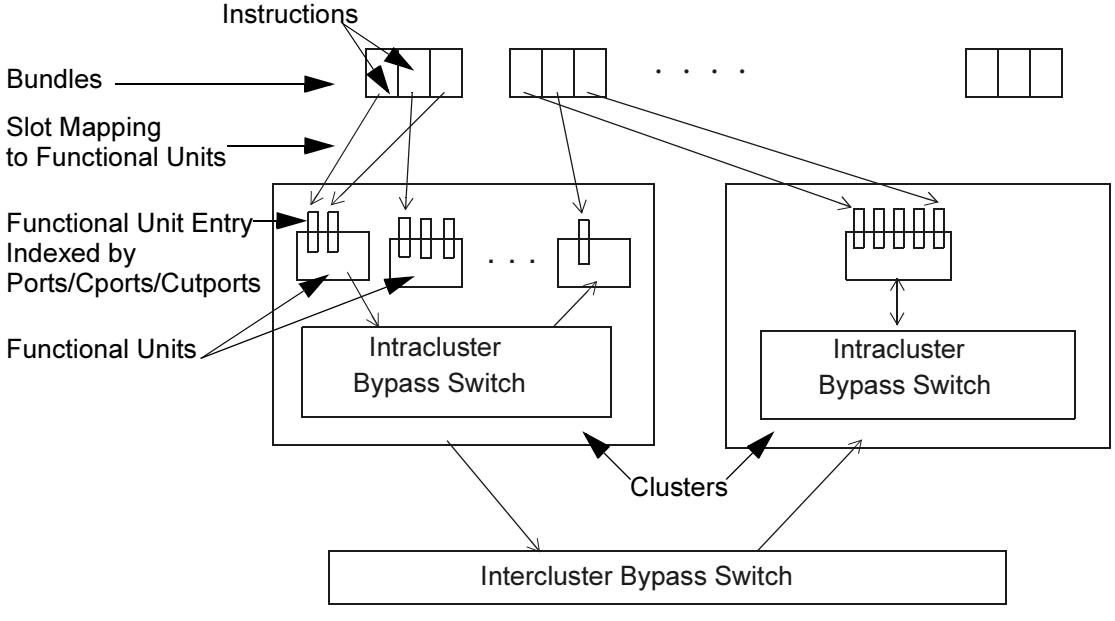
Abstract Microarchitecture Model

The model used by KAPI to represent potential Itanium architecture designs includes, but is not limited to the following:

- issue width
- functional unit latencies
- instruction slot to functional unit mapping (dispersal or expansion)
- clustered functional units
- number and type of functional units
- available bypasses
- cache configuration and size

The general microarchitectural model is shown in Figure 4-1. Each bundle is composed of three instruction slots. The instruction in each instruction slot is mapped to a functional unit for execution. Functional units may be divided into several clusters, which might increase latency when the result of an operation is needed outside the cluster.

Figure 4-1 Abstract Microarchitectural Model



Knobsfile Configurability

NOTE. *Some tools can more easily reconfigure their behaviour to match different microarchitectural features than other tools.*

KAPI is highly configurable, even though not every tool can take advantage of such flexibility. This flexibility implies that tools must provide some form of 'self validation' to make sure that they can handle the knobable values passed. Additionally,

the knobsfile author is responsible for keeping various knobs 'semantically consistent' by not setting different knobs to have contradictory values. Adjusting knobsfiles requires some expertise.

Accessing Intel® Itanium™ Architecture Information 5

This chapter describes the specific conventions and header files needed to access Intel® Itanium™ architecture information in KAPI. It examines the instruction ID, source and destination registers, and Itanium architecture-specific interface types.

Itanium Architecture Usage Models

This section explains the two usage models for the Itanium architecture layer.

Model 1

The first model uses KAPI to parse the knobsfile(s), initialize the Itanium architecture layer, and then use the Itanium architecture functions.

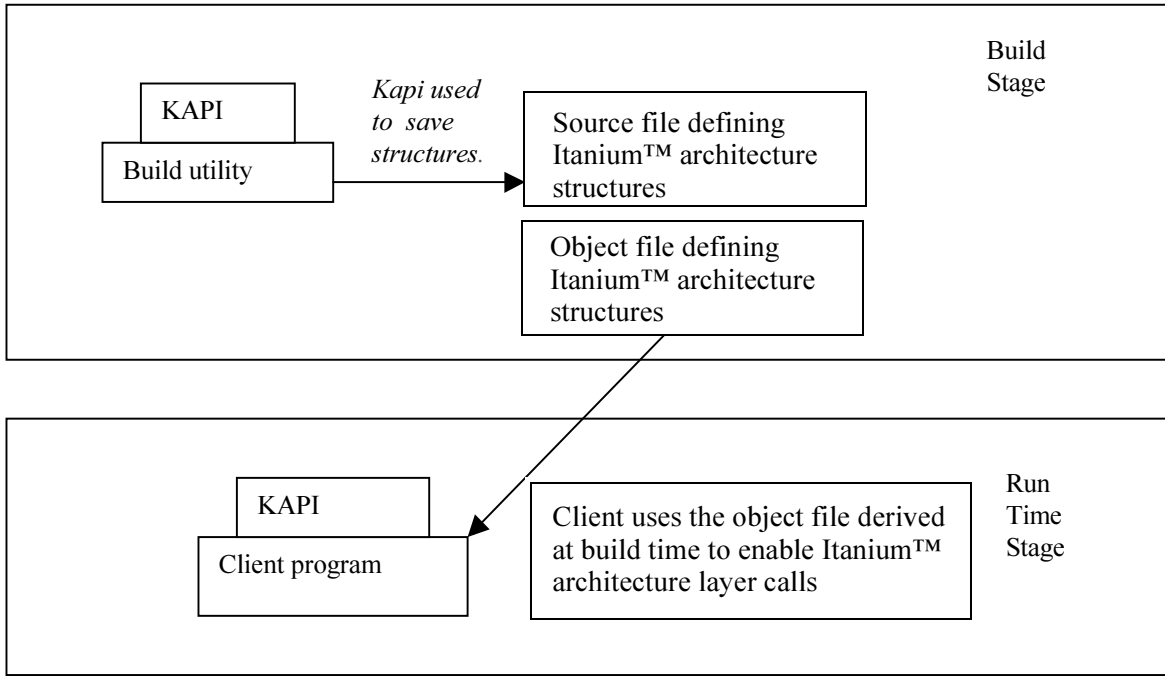
The input is text from the knobsfile(s).

The Intel C++ Compiler uses this model at the time at which the compiler is built, rather than when the compiler is run.

Model 2

The second model uses the parsed C code from the KAPI layer as input, incorporating the benefits of using the Itanium architecture layer functions provided by KAPI during run time, without paying the penalty of parsing the knobsfile, and initializing the Itanium architecture layer structures. See Figure 5-1.

Figure 5-1 Model 2



This model permits the use of Itanium architecture-specific functions with internal logic. These functions are difficult to represent in tables, and difficult to use at build time; for example, when a function maps an instruction according to Merced mapping rules.

Using a special set of functions described in Saving Information in Header Format Functions section in Chapter 7 or a utility supplied by a KAPI owner interfacing with these functions, the client parses the knobsfiles, builds a special source file and compiles it at build time. This file is then linked with the rest of the project for initializing the Itanium architecture layer structures or part of these structures.

The client initializes the Itanium architecture layer by passing a pointer to the created structures, and can thereafter use the Itanium architecture functions related to these structures

Unlike the first model, when this model is used at run time, the Itanium architecture layer cannot be initialized multiple times with different values. The knobsfile is parsed during build time.

Primary Instruction Set Issues

In general, to provide an interface to performance characteristics of the Itanium instruction set, all tools use the following:

- A unique number for each Itanium architecture operation, called an instruction ID.
- A way of identifying and describing the source and destination operands of instructions.

Individual tools are likely to internally represent instructions differently. Thus, all tools using KAPI must provide a translation for the fixed mapping that is chosen by KAPI.

Instruction ID

Use the `inst_ids.h` file, which provides a complete list of Itanium instructions in the form of an enumerated type. The instruction codes are suitable for indexing arrays, except for index `0` that has two dummy op-codes mapped to it.

By convention, the instruction IDs are represented by the KAPI integer type:

```
typedef int kapi_iid_t;
```

This type is provided rather than using the type `Inst_id_t` that is defined in the specified header file. All of the values defined by `Inst_id_t` are identical to those for `kapi_iid_t`.

Source/Destination Operands

KAPI defines the latency between functional unit classes (`fuS`). Translating an instruction operand to a functional unit class operand is trivial, but when the instruction has more than one destination (or more than one source), you must inform KAPI which is the intended operand. Therefore, a functional unit class' latency may change in accordance with the operand.

You must specify source and destination operands since some latencies depend on the operand produced (e.g., a loaded value or the post-increment register), or the operand that uses it (e.g., the address value or data value in a store instruction).

KAPI uses a small set of mnemonics to name operands. KAPI uses functions to map those mnemonics to corresponding integer values. The names are determined at runtime.

Generally, the latency-related functions assume a primary operand when no operand name is provided. For example, functional unit classes `fuICMP` and `fuFCMP` assume a predicate register as the primary destination operand. However, since some instructions have no explicit operands (such as stores, which normally do not write registers), a special `DUMMY` destination operand is listed as the primary destination operand for such instructions.

Specifically, for each instruction you can use the function `KAPI_GetOppIndex()` to determine the KAPI operand type for each explicit operand, using the operand order that appears in the *Intel® Itanium™ Architecture Software Developer's Manual*.

The functional unit class names are described below:

- The load instructions belong to the `fuLFETCH`, `fuFLD`, `fuFLDP`, `fuCLD`, and `fuFCLD` functional unit classes.
- The store instructions belong to the `fuST` and `fuSTF` functional unit classes.

NOTE. *The tables below outline the operand types for the knobsfiles released with KAPI version 4.1. However, the operand names mentioned in the tables are configurable in the knobsfile, so you should use the functions to determine KAPI operand names.*

Table 5-1 lists the operands for all predicatable instructions. Table 5-2 through Table 5-6 describe special cases.

Table 5-1 All Predicatable Instructions

	Operand in Use	Referenced Operand
sources	*predicate*	qualifying predicate
	primary	primary source operand(s)

Table 5-2 cmp, fcmp, tbit, tnat Instructions

	Operand in Use	Referenced Operand
sources	*predicate*	qualifying predicate
	primary	primary source operand(s)
destinations	*predicate*	the predicate being set
	primary or unspecified	*predicate*

Table 5-3 Store Instructions

	Operand in Use	Referenced Operand
sources	*addr*	address register
	data	data register being stored
	predicate	qualifying predicate
	primary or unspecified	*data*

Table 5-3 Store Instructions (continued)

	Operand in Use	Referenced Operand
destinations	*addr*	for post-increment versions
	ar_unat	for spill version
	DUMMY or *primary* or unspecified	*no destination*

Table 5-4 Load-type Instructions

	Operand in Use	Referenced Operand
sources	*addr*	address register
	incr	post increment register
	predicate	qualifying predicate
	primary or unspecified	*addr*
destinations	*data*	value loaded register
	addr	address register (for post-increment)
	primary or unspecified	*data* (except LFETCH which has DUMMY as its primary)

Table 5-5 xchg, cmpxchg, and fetchadd Instructions

	Operand in Use	Referenced Operand
sources	*addr*	address register
	xchg	exchange register
	predicate	qualifying predicate
	ar_ccv	ar.ccv
	primary or unspecified	*addr*
destinations	*data*	value loaded register
	primary or unspecified	*data*

Table 5-6 fprcpa, fprsqrta, frcpa, and frsqrta Instructions

	Operand in Use	Referenced Operand
sources	*data*	registers that supply the input floating point values to the instructions
	predicate	qualifying predicate
	primary or unspecified	*data*
destinations	*data*	destination floating-point register to which this instruction writes
	predicate	predicate register
	primary or unspecified	*data*

Table 5-7 FRAR_M/TOAR_M

	Operand in Use	Referenced Operand
sources (FRAR_M)	*ar_k0, ..., ar_k7, ar_rsc, ar_bsp, ar_bspstore, ar_rnat, ar_fcr, ar_eflag, ar_csd, ar_ssd, ar_cflg, ar_fsr, ar_fir, ar_fdr, ar_ccv, ar_unat, ar_fpsr, ar_itc*	corresponding application register
	predicate	qualifying predicate

Table 5-7 FRAR_M/TOAR_M (continued)

	Operand in Use	Referenced Operand
	primary or unspecified or DUMMY	*No default sources*
destinations (TOAR_M)	*ar_k0, ..., ar_k7, ar_rsc, ar_bspstore, ar_rnat, ar_fcr, ar_eflag, ar_csd, ar_ssd, ar_cflg, ar_fsr, ar_fir, ar_fdr, ar_ccv, ar_unat, ar_fpsr, ar_itc*	corresponding application register
	predicate	predicate register
	primary or unspecified or DUMMY	*No default destinations*

Table 5-8 FRAR_I/TOAR_I

	Operand in Use	Referenced Operand
sources (FRAR_I)	*ar_pfs, ar_lc, ar_ec*	corresponding application register
	predicate	qualifying predicate
	primary or unspecified or DUMMY	*No default source*
destinations (TOAR_I)	*ar_pfs, ar_lc, ar_ec*	corresponding application register
	predicate	predicate register
	primary or unspecified or DUMMY	*No default destinations*

Table 5-9 FRCR/TOCR

	Operand in Use	Referenced Operand
sources (FRCR	*cr_dcr, cr_itm, cr_iva, cr_pta, cr_ipsr, cr_isr, cr_iip, cr_ifa, cr_itir, cr_iipa, cr_ifs, cr_iim, cr_iha, cr_lid, cr_ivr, cr_tpr, cr_eoi, cr_irr0, ..., cr_irr3, cr_itv, cr_pmv, cr_cmcv, cr_lrr0, cr_lrr1*	corresponding control register
	predicate	qualifying predicate
	primary or unspecified or DUMMY	*No default sources*
destinations (TOCR)	*cr_dcr, cr_itm, cr_iva, cr_pta, cr_ipsr, cr_isr, cr_iip, cr_ifa, cr_itir, cr_iipa, cr_ifs, cr_iim, cr_iha, cr_lid, cr_tpr, cr_eoi, cr_itv, cr_pmv, cr_cmcv, cr_lrr0, cr_lrr1*	corresponding control register
	predicate	predicate register
	primary or unspecified or DUMMY	*No default destinations*

Use the functions `KAPI_oppGetSource()` and `KAPI_oppGetDest()` to convert these names to KAPI-specific integers that are passed to functions. The value zero represents the primary source and primary destination operands. No operand name lookup is required to find the corresponding integer value when a primary operand exists because zero is always used for primary operands. The Itanium architecture interface functions exclusively use the integer representation of operands. The knobsfile specifies operand-related attributes as follows:

- primary latencies in the `fuLatency` array variable
- other operand latencies using the `LATENCY` attribute

- operand names using the `SOURCES` and `DESTINATIONS` attributes

NOTE. *When specifying latencies and bypasses, specify both the producer operand and consumer operand along with the relevant functional unit classes. If the operands are omitted, the primary source or destination is assumed.*

Basic Itanium Architecture-specific Interface Types

This section describes a structured interface that allows you to query some characteristics of the instruction set, functional units, and bypasses. While you can query other machine attributes, the interface is unstructured.

The execution units in a given Itanium microarchitecture are identified by integers of these types:

```
typedef int kapi_port_t;
typedef int kapi_cport_t;
typedef int kapi_cutport_t;
```

Each port (functional unit) in the machine is associated with one of the following instruction slot types:

`I`	immediate
`M`	memory
`F`	floating-point
`B`	branch
`L`	long

The `cport` type is used for specifying ports relative to a specific cluster. The `port` type gives each port a unique number. Thus, it is possible for two or more clusters to each have a `cport` numbered `0`, but there is only one `port 0` in the entire machine. The mapping of `port`s and `cport`s is not statically defined. KAPI may choose any convenient numbering,

so you must use the KAPI_cutportInfo() function to extract a cport number from a cluster, unit type, and unit number. For example, cportM0 would be unit type kapi_utM and unit number 0.

Cluster unit type port (cutports) represent the number of a specific unit of a given type. For example, cportF3 would be cutport 3 of type kapi_utF. Use cutport numbers to refer to ports that are not specific to a given cluster.

cport numbers are well-defined only when combined with a cluster number.

Ports are grouped into clusters to simplify bypass matrix specification and to provide a more structured model for compilers. Clusters are identified by integers from 0 to the number of clusters minus one.

```
typedef int kapi_cluster_t;
```

Since the set of unit types does not exactly correspond to the instruction slot types (there are no L units), the ut_t type below is used to describe types of functional units:

```
typedef enum KAPI_UT_T_ {
    kapi_utM,
    kapi_utI,
    kapi_utB,
    kapi_utF
} kapi_ut_t;              /* unit type */
```

To specify the behavior of the functional units with respect to clustering, latency, and bypass characteristics, the knobsfile defines a set of functional unit classes (identified with the integer type kapi_fu_t):

```
typedef int kapi_fu_t;
```

The execution, latency, and bypass characteristics of each functional unit class are defined in the knobsfile in terms of functional units and clusters. Once the characteristics of each functional unit class are defined, we associate a functional unit class with each instruction to define its latency characteristics and the set of functional units on which it can execute.

Finally, given that instructions have been classified and named, and functional units have been classified and named, the knobsfile allows you to specify the set of instructions that a given functional unit class can execute. List the kapi_fu_t set that each functional unit

can perform, thereby providing the final piece that links functional units, functional unit classes, and instructions. The next section describes how to query latency and bypass information based on `iid`, `port`, and `fu`.

KAPI also allows some architectural information to be queried and configured even though it is unlikely that such classifications will change or that tools will seamlessly support such configurability. The most useful information follows:

instruction type	queries the type and position of instruction slots into which an individual instruction can go
bundle identification numbers	queries bundle resources such as availability and retrieval of instruction slots
instruction slot types	lists the architecturally defined instruction slot names and retrieves information on specific instruction slot types to be accessed

Instruction types are defined by

```
typedef enum KAPI_IT_T {
    kapi_itA,
    kapi_itM,
    kapi_itI,
    kapi_itB,
    kapi_itBl,
    kapi_itF,
    kapi_itL
} kapi_it_t;
```

Bundle identification numbers are given by

```
typedef int kapi_bid_t;
```

Instruction slot types are given by

```
typedef KAPI_SYL_T {
    kapi_sylI,
    kapi_sylM,
    kapi_sylF,
```

```
    kapi_sylB,
    kapi_sylL
} kapi_syl_t;
```

For convenience, each of these types has a FIRST, LAST, and n (number of) definitions that allow for easy array allocation and indexing. Assume that the number of values of each type is one greater than the LAST value of each type, and that the range of values used is a small, nearly dense space. That is, it is reasonable to declare arrays of length nSYL.

```
#define kapi_sylFIRST
#define kapi_sylLAST
#define kapi_nSYL           /* number of instruction slot types */
#define kapi_itFIRST
#define kapi_itLAST
#define kapi_nIT            /* number of instruction types */
#define kapi_bidFIRST
#define kapi_bidLAST
#define kapi_nBID           /* number of bundle IDs (16) */
#define kapi_nUT            /* number of unit types */
#define kapi_utFIRST
#define kapi_utLAST
```

Defining Intel® Itanium™ Microarchitecture

6

This chapter describes variables and attributes that define Intel® Itanium™ microarchitectures and upon which all of the Itanium specific functions depend. These values are required by the Itanium architecture-specific layer in KAPI. Note that extensions to include clustering have made it desirable for the Itanium architecture mode to allow for some violations of KAPI's strict typing conventions. These extensions are described later. Any tool that uses the KAPI Itanium architecture API to get information about the microarchitecture must adhere to the knobs' descriptions. This API does not dictate the meaning or content of any other knobs. Tools can add new knobs, provided they correctly support those described here.

Not all tools support all possible values of all possible knobs and types. This API only provides a mechanism for specifying values. It does not imply that any given tool (simulators, compilers, performance tools, etc.) can in fact accommodate the level of flexibility that KAPI and the knobsfile provide.

Do not confuse the types and variables defined in the knobsfile with the names of types and variables defined in the C header files, although there is an obvious relationship between the two.

Types

All of the following types must be defined in the knobsfile for the Itanium architecture interface to be used.

cache_names_t

The `cache_names_t` type defines the names of all the caches in the memory hierarchy that will be used in the knobsfile. The convention for usage is that the I and D hierarchies are in a tree structure with separate I and D caches beyond which there is a hierarchy of unified (U) caches. The relative level and type of content for each cache is specified by the `CACHE_RelativeLevel` and `CACHE_Content` variables.

```
TYPE cache_names_t = enum ( L1I, L1D, L2, L3 );
```

While these may be redefined, some tools may depend on a specific hierarchy.

cache_content_t

The `cache_content_t` type defines the types of content that a cache can contain. This is used by the `CACHE_Content` variable.

```
TYPE cache_content_t = enum ( content_other, content_instruction,
    content_data );
```

New values in the enumerated type may be added, but only at the end of the current type definition. The tools depend on the order and names of the current enumeration identifiers.

cache_policy_write_t

The `cache_policy_write_t` type defines the write policies that a cache can use. This is used by the `CACHE_PolicyWrite` variable. 'wb' is write back, 'wt' is write through.

```
TYPE cache_policy_write_t = enum ( policy_write_other,
     policy_write_wb, policy_write_wt );
```

New values in the enumerated type may be added, but only at the end of the current type definition. The tools depend on the order and names of the current enumeration identifiers.

cache_policy_repl_t

The `cache_policy_repl_t` type defines the replacement policies that a cache can use. This is used by the `CACHE_PolicyRepl` variable.

Values:

`lru` least recently used

`nru` not recently used

```
TYPE cache_policy_repl_t = enum ( policy_repl_other, policy_repl_lru,
     policy_repl_nru );
```

New values in the enumerated type may be added, but only at the end of the current type definition. The tools depend on the order and names of the current enumeration identifiers.

cache_policy_alloc_t

The `cache_policy_alloc_t` type defines the allocation policies that a cache can use. This is used by the `CACHE_PolicyAlloc` variable.

Values:

`wa` write allocate

`nwa` no write allocate

```
TYPE cache_policy_alloc_t = enum ( policy_alloc_other,
     policy_alloc_wa, policy_alloc_nwa );
```

New values in the enumerated type may be added, but only at the end of the current type definition. The tools depend on the order and names of the current enumeration identifiers.

cache_port_access_t

The `cache_port_access_t` type defines the types of access that a given port can support. This is used by the `CACHE_PortXAccessTypes` variable.

Values:

`read` memory reads

`write` memory write accesses

`snoop` snoops from higher levels in the hierarchy

`prefetch` prefetch requests

```
TYPE cache_port_access_t = enum  ( access_other, access_read,
     access_write, access_snoop, access_prefetch );
```

New values in the enumerated type may be added, but only at the end of the current type definition. The tools depend on the order and names of the current enumeration identifiers.

ut_t

The `ut_t` type represents the four basic types of execution units found in current Itanium architecture designs. This type cannot be changed.

```
TYPE ut_t = enum ( utM, utI, utF, utB )
limit <> noredefine;
```

To maintain compatibility, the order of the unit types specified in the definition is enforced and results in a syntax error if changed.

syl_t

The `syl_t` type represents the architecturally defined instruction slot types of which bundles are composed. This type cannot be changed.

```
TYPE syl_t = enum ( sylI, sylM, sylF, sylB, sylL )
limit <> noredefine;
```

bid_t

This is a list of all architecturally defined bundle types. The `bid_t` type cannot be changed. It is as follows:

```
TYPE bid_t = enum ( bidMII, bidMI_I, bidMLX, bidRESERVED_3, bidMMI,
    bidM_MI, bidMFI, bidMMF, bidMIB, bidMBB, bidRESERVED_A, bidBBB,
    bidMMB, bidRESERVED_D, bidMFB, bidRESERVED_F )
limit <> noredefine;
```

The definition of `bid_t` does not strictly follow Itanium architecture specifications for bundle templates.

Itanium architecture defines twice as many bundles as provided by KAPI. However, for ease of use, KAPI combines bundle types that have the same slot structure, but that differ only by whether the bundle has a stop after the last slot or not. For example, the Itanium architecture specification has both an `MII` and `MII_` bundle template, but KAPI only provides `MII`, with the implicit understanding that there are two versions: `MII` and `MII_`.

it_t

This is the list of all instruction types as taken from EMDB. The `it_t` type cannot be redefined. It is defined as follows:

```
TYPE it_t = enum ( itA, itM, itB, itBl, itI, itF, itL )
limit <> noredefine;
```

fu_t

The `fu_t` type lists a set of instruction classes that are equivalent under bypass and latency characteristics.

The Itanium architecture functions in KAPI depend on the following `fu_t` enumerated type names and their memory operations: `fuLD`, `fuST`, `fuFLD`, `fuFLDP`, `fCLD`, and `fuFCLD`. Additionally, since these operations have multiple source and/or destination operands, they cannot be omitted or removed or have their memory operations changed without causing errors in the interpretation of multiple source/destination instructions.

NOTE. *The definition of fu_t is likely to change between knobsfile releases and especially for knobsfiles from one microprocessor to another. Except where noted above, tools should not depend on the order, names, number, or interpretation of fu_t class names for correct operation of their tool. If tools do so, they may need revision for each knobsfile. This note does not change the semantics of the API, but is intended to provide guidance to users of the API.*

fu_info_t

The `fu_info_t` type defines attributes that the knobsfile assigns to functional unit classes. The following attributes have related API functions and should not be removed:

`fu_info_approximate_latency` Functional unit classes with this attribute have only approximate latency information. This is generally due to the complexity of defining latency for those classes.

`fu_info_no_latency_info` Functional unit classes with this attribute have no latency information.

operand_direction_t

The `operand_direction_t` type defines attributes that the knobsfile assigns to functional unit classes. The following attributes have related API functions and should not be removed:

`operand_direction_source` The accessed source operand qualifies the latency of functional unit classes with this attribute. In most cases, the destination operand is used for this purpose, but for application registers (ARs) and control registers (CRs) this approach correctly represents the latencies for the move-to and move-from instructions.

`operand_direction_destination` The accessed destination operand qualifies the latency of functional unit classes with this attribute. This is the normal case for most functional unit classes.

See the "operand_direction : array[fu_t] of operand_direction_t;" section for more information on usage of the `operand-direction_t` type.

Variables

See "Instruction Latency Functions" in Chapter 7 before reading the descriptions of these variables.

fuInfoBits: array[fu_t] of bitmask(fu_info_t);

The `fuInfoBits` variable associates `fu_info_t` bits with functional units. For example:

```
fuInfoBits[ fuSYST_M ] := bitmask(fu_info_approximate_latency);
fuInfoBits[ fuSYST_M0 ] := bitmask(fu_info_approximate_latency);
```

fuLatency : array[fu_t] of integer;

The `fuLatency` variable defines the primary destination operand's core latency. This is the number of cycles required before an instruction consuming the result register of an instruction of type `fu_t` is available to a consumer instruction. This does not include bypass time.

For instructions with multiple result registers, see the discussion of the `LATENCY` attribute. By default, every `fu_t` has a primary `opp`.

The values returned by the function `KAPI_CoreLatency()` are based partially on this variable. See "Instruction Latency Functions" in Chapter 7 to interpret the values of this variable.

NumberOfClusters: integer;

The `NumberOfClusters` variable sets the number of clusters. Clusters are referenced by a number from `0` to `(NumberOfClusters-1)`. There must be at least one cluster.

The value is `1` by default.

NOTE. *In functions that require a cluster number argument, cluster numbers greater or equal to the value of* `NumberOfClusters` *may cause error conditions to be returned.*

cluster*X*MaxBundleIssue : integer;

The `cluster`*`X`*`MaxBundleIssue` variable sets the number of bundles that are issued to cluster *`X`*. This model assumes that the first `cluster0MaxBundleIssue` bundles are sent to cluster 0, the next `cluster1MaxBundleIssue` bundles are sent to cluster 1, and so on.

cluster*X*Cutports : array[ut_t] of integer;

The `cluster`*`X`*`Cutports` variable sets the number of functional units of each unit type (`ut_t`) for cluster *`X`*. Values for *`X`* are `0`, `1`, ..., `NumberOfClusters-1`.

Setting this variable implicitly declares `cport` specifiers (strings of the format) which are used to index the `cluster`*`X`*`CportMask` variable. The following example indicates that cluster 0 has two I-units, and implicitly defines the strings `cportI0` and `cportI1` to index `cluster`*`X`*`CportMask`:

```
cluster0Cutports[ utI ] := 2;
```

The following definition allows for the use of `cportI0`, `cportI1`, `cportI2`, and `cportI3` in contexts where `cport_spec`'s are required for the specified cluster.

```
cluster1Cutports[ utI ] := 4;
```

cluster*X*CportMask : array[string] of bitmask(fu_t);

The `cluster`*`X`*`CportMask` variable is an associative array indexed by `cport`*`XY`* strings. It specifies the kinds of instructions that a given port can execute, as given by a bitmask of functional unit class specifiers.

By convention these index values are `cport`*`YZ`* where *`Y`* is one of I, M, B, or F and *`Z`* is a number between 0 and the number of *`Y`*-type ports minus one. These index values are called local port specifiers and are implicitly declared when the values of the `cluster`*`X`*`Cports` variables are defined. Thus, `cluster`*`X`*`Cports` must be defined before `cluster`*`X`*`CportMask` for any given value of *`X`*.

operand_direction : array[fu_t] of operand_direction_t;

The `operand_direction` array variable defines whether latencies for a given functional unit class are qualified by which source operand is accessed or which destination operand is accessed. For the current reference knobsfile, in all cases except `FRCR`, `FRAR_M`, and `FRAR_I`, all functional units use `operand_direction_destination`.

CACHE_Content : array[cache_names_t] of bitmask (cache_content_t);

The `CACHE_Content` variable specifies the kinds of data that may be stored in the indicated cache. If both the I and D cache content bits are set, then the cache is assumed to be a U cache.

CACHE_RelativeLevel : array[cache_names_t] of cache_relative_level_t;

The `CACHE_RelativeLevel` variable specifies the relative level of the indicated cache within the cache hierarchy. For example, the L0 I cache is 0, the L1 I cache is 1, the L0 D cache is 0, and the L1 D cache is 1. The first level of unified cache is 0, etc.

CACHE_PolicyWrite : array[cache_names_t] of cache_policy_write_t;

The `CACHE_PolicyWrite` variable specifies the write policy of the indicated cache.

CACHE_PolicyRepl : array[cache_names_t] of cache_policy_repl_t;

The `CACHE_PolicyRepl` variable specifies the replacement policy of the indicated cache.

CACHE_PolicyAlloc : array[cache_names_t] of cache_policy alloc_t;

The `CACHE_PolicyAlloc` variable specifies the allocation policy of the indicated cache.

CACHE_Lines : array[cache_names_t] of int;

The `CACHE_Lines` variable specifies the number of lines in the indicated cache.

CACHE_BytesPerLine : array[cache_names_t] of int;

The `CACHE_BytesPerLine` variable specifies the number of bytes in each line of the indicated cache.

CACHE_Ways : array[cache_names_t] of int;

The `CACHE_Ways` variable specifies the number of ways in the indicated cache.

CACHE_ReadLatency : array[cache_names_t] of int;

The `CACHE_ReadLatency` variable specifies the best case latency of an integer load to the specified cache.

CACHE_Ports : array[cache_names_t] of int;

The `CACHE_Ports` variable specifies the number of simultaneous accesses in the indicated cache. The types of those accesses are provided in the `CACHE_PortXAcessTypes` variable.

CACHE_Port*X*AccessTypes : array[cache_names_t] of bitmask (cache_port_access_t);

The `CACHE_Port`*X*`AccessTypes` variable specifies a bitmask of type of accesses the port specified by *X* can have in the indicated cache.

Attributes

This section describes the KAPI attributes.

SOURCES

The `SOURCES` attribute defines valid operand names for each functional unit class. By default, the knobsfile defines a `primary` source if the functional unit class does not have an associated `SOURCES` attribute. An asterisk (`*`) within a `SOURCES` attribute indicates a primary source.

By default, all classes have a `predicate` source operand representing the qualifying predicate.

Not all variants of all instructions associated with a functional unit class include the specified operand(s).

For example:

```
SOURCES += "fuLD:predicate,*addr,incr";
SOURCES += "fuFLD:predicate,*addr,incr";
```

DESTINATIONS

The `DESTINATIONS` attribute defines valid operand names for each functional unit class. By default, the knobsfile defines a `primary` destination if the functional unit class does not have an associated `DESTINATIONS` attribute. An asterisk (`*`) within a `DESTINATIONS` attribute indicates a primary destination.

Not all variants of all instructions associated with a functional unit class include the specified operand(s).

For example:

```
DESTINATIONS += "fuLFETCH:addr";
DESTINATIONS += "fuSEM:*data";
```

LATENCY

The LATENCY attribute lists secondary destination latencies for functional unit classes. The format of the expression is:

```
LATENCY += "fu/operandName=X"
```

operandName is defined for only those functional unit classes and secondary names given in 7.1 Usage Models. Both fu and operandName must be specified.

INTRACLUSTER

The INTRACLUSTER attribute specifies intra-cluster bypassing for all clusters or one specific cluster. The format of intra-cluster bypass specifications is:

```
INTRACLUSTER += "{cluster\}fu1{/on1{/cport1}}
    :fu2{/on2{/cport2}}=X"
```

Where cluster is the name of a valid cluster, fu1 and fu2 are names of valid functional unit classes, cport1 and cport2 are names of valid functional units (cports), both of which must reside in cluster. on1 and on2 are optional operand name specifiers for source and destination registers, when required.

cluster\ is optional. If omitted, the bypass applies to any cluster. If specified, it only affects the cluster in question. Note that the separator between the cluster name and the first fu is a backslash and not a forward slash.

fu1 and fu2 must always be specified and fu1 is the functional unit class of the instruction producing a result that is consumed by an instruction of functional unit class fu2.

If on1 and on2 fields are omitted, the primary source and destination operands are implied. The on1 and on2 fields can only be omitted if the port specifications are also omitted.

The port specifications are the same names that are of the form cport*XY*. Values for *X* are I, M, B, or F. Values for *Y* are integers from 0 to the number of ports of that type available on a cluster.

If any of the port or cluster specifiers are invalid (whether undefined, or a specific port isn't present on the specified cluster), then the entire entry is ignored during lookups.

These entries are used to construct lookup data for the function `KAPI_IntraClusterBypass()`. During lookup, entries are searched in reverse order (later entries override earlier ones) and the first matching entry for the search is returned.

INTERCLUSTER

The `INTERCLUSTER` attribute specifies inter-cluster bypassing for pairs of clusters. The format of inter-cluster bypass specifications is:

```
INTERCLUSTER += "clusterX/fu1{/on1{/cport1}}
                :clusterY/fu2{/on2{/cport2}}=Z"
```

Since this format is for intercluster latencies, `clusterX` and `clusterY` are the names of source and destination clusters. `X` and `Y` are integers. They are smaller than `NumberOfClusters`.

`fu1` and `fu2` are names of valid functional unit classes.

`cport1` and `cport2` are names of valid `cports`.

`on1` and `on2` are operand name specifiers for source and destination registers.

`clusterX and clusterY` must be different clusters since this attribute is for intercluster bypass specification.

`fu1` and `fu2` must always be specified and `fu1` is the functional unit class of the instruction producing a result that is consumed by an instruction of functional unit class `fu2`.

The `on1` and `on2` fields can be omitted if no port specifiers are listed. If omitted, then `primary` is assumed.

The port specifications are the same as the names of the form `cport`*`XY`*. Values for *`X`* are I, M, B, or F. Values for `Y` are from `0` to the number of ports of that type available on a cluster.

If any of the port or cluster specifiers are invalid (whether not defined, or a specific port is not present on the specified cluster), then the entire entry is ignored during lookups.

CLUSTERDISTANCE

The `CLUSTERDISTANCE` attribute specifies the fixed inter-cluster component of bypass latency. Setting these values allows a concise specification of the cycle penalty for transferring data between clusters. The format is:

```
CLUSTERDISTANCE +=
     "clusterX:clusterY=2"
```

These entries look up data for calls to the function `KAPI_ClusterDistance()`. During lookup, entries are searched in reverse order (later entries override earlier ones) and the first matching entry for the search is returned. *X* and *Y* are integers. They are smaller than the number of clusters.

BYPASS

The `BYPASS` attribute is an unsupported KAPI 2.x feature. If this attribute appears in a KAPI 4.x knobsfile, an error is issued.

Instruction

The `instruction` attribute specifies the instruction type and instruction ID (`iid`) to mnemonic mappings for the Itanium instruction set. It associates new `fu_t` and `it_t` with instructions. It also associates the operand index as indicated in the *Intel® Itanium™ Architecture Software Developer's Manual* with the KAPI operand type for the instruction.

Instruction Format

```
instruction += "iid,emid,mnem,it,fu:exp_ops:imp_ops"
```

where

```
iid           KAPI instruction ID
emid          unique mnemonic
mnem          instruction mnemonic
it            instruction type
fu            functional unit class
exp_ops       explicit operand list
```

`imp_ops` implicit operand list

Operand List Syntax

`<op specifier>[*]<name>` (repeat for operand role) `/` (repeat for operand type)

where

`<op specifier>` May be one of

`?` for qualifying predicate

`+` for destination

`-` for source

`[*]` Signifies a primary operand. This parameter is optional as it matches the definition in `SOURCES` and `DESTINATIONS` attributes.

`<name>` This is the KAPI operand name.

Examples

```
instruction += "533,EM_LD4_S_NT1_R1_R3_R2,ld4.s.nt1,itM,fuLD:
    ?predicate,+data,+addr-addr,-incr:none";
instruction += "1249,EM_MOV_M_AR3_R2,mov.m,itM,fuTOAR_M:
    ?predicate,+ar_rsc/+ar_bsp/+ar_pfs/+ar_lc/+ar_ec,-primary:none";
instruction += "1254,EM_ALLOC_R1_AR_PFS_I_L_O_R,alloc,itM,fuSYST_M0:
    ?predicate,+primary,-ar_pfs:none";
```

KAPI Functions

7

This chapter lists the basic Intel® Itanium™ architecture-specific information functions. The functions depend on specific variables and types defined in the input knobsfile. If the variables are not defined, then the information for these functions is not available. To use these functions, `KAPI_ia64_Initialize()` must be called.

If `KAPI_ia64_Initialize()` returns true, the structured information is available and the functions in this chapter can be called. If an error is returned, some information is missing, error messages are printed, and the information returned by these functions is unpredictable and may cause program faults.

Unless otherwise specified, if the functions are supplied with invalid input arguments, the return values of the functions in this chapter are undefined and may cause a program fault.

The functions are grouped as follows:

- instruction set queries
- operand
- ports and clustering
- instruction latencies
- Itanium architecture bundle content queries
- cache hierarchies
- saving information in header format

Instruction Set Query Functions

These functions enable retrieval of information regarding instructions.

KAPI_iidCount( )

Returns the number of instructions

```
int KAPI_iidCount( void *pConfig )
```

Parameters

pConfig The pointer intialized by `KAPI_init()`.

Description

The `KAPI_iidCount()` query returns the number of instructions (`iid`'s).

KAPI_iid2it( )

Returns the instruction type of the corresponding instruction code

```
kapi_it_t KAPI_iid2it( void *pConfig, kapi_iid_t iid, int iReserved )
```

Parameters

pConfig	The pointer intialized by KAPI_init().
iid	Instruction ID.
iReserved	Reserved for extensions and future microarchitectures. Use 0.

Description

Given an *iid*, the KAPI_iid2it() query returns the instruction type of the corresponding instruction code. The instruction type describes the instruction slot restrictions of the instruction (itA, itM, itI, itB, itBl, itF, itL) as given in the knobsfile.

KAPI_iid2fu( )

Allows lookup of a given instruction's fu

```
kapi_fu_t KAPI_iid2fu( void *pConfig, kapi_iid_t iid, int iReserved )
```

Parameters

pConfig	The pointer initialized by `KAPI_init()`.
iid	Instruction ID.
iReserved	Reserved for extensions and future microarchitectures. Use `0`.

Description

The `KAPI_iid2fu()` function looks up the given instruction's functional unit class based on its *iid* and possibly an offset value (where accessing numbered application registers).

KAPI_iid2mnemonic( )

Allows lookup of a string representation of an instruction

```
char *KAPI_iid2mnemonic( void *pConfig, kapi_iid_t iid, int iReserved)
```

Parameters

pConfig	The pointer intialized by KAPI_init().
iid	Instruction ID.
iReserved	Reserved for extensions and future microarchitectures. Use 0.

Description

The KAPI_iid2mnemonic() function looks up a string representation of an instruction based on its *iid* and possibly an offset value in the case of access to numbered application registers. The string returned is not necessarily the one that is needed by the assembler. The string is provided mainly for debugging purposes. The name returned is not necessarily unique (see "KAPI_iid2uniqueName( )" below).

KAPI_iid2uniqueName( )

Allows lookup of a string representation of an instruction

```
char *KAPI_iid2uniqueName( void *pConfig, kapi_iid_t iid,
int iReserved )
```

Parameters

pConfig	The pointer intialized by `KAPI_init()`.
iid	Instruction ID.
iReserved	Reserved for extensions and future microarchitectures. Use `0`.

Description

The `KAPI_iid2uniqueName()` function looks up the string representation of an instruction based on its *iid* and possibly an offset value in the case of access to numbered application registers. The string returned is not the one that is needed by the assembler. The string is provided mainly for debugging purposes. The default names are based on the enumerated type header files released for the assembler tools.

KAPI_uniqueName2iid( )

Allows lookup of a numeric representation of an instruction

```
kapi_iid_t KAPI_uniqueName2iid( void *pConfig, char *pchName,
int iReserved )
```

Parameters

pConfig	The pointer intialized by `KAPI_init()`.
pchName	String representation of `inst_id_t` of instruction ID.
iReserved	Reserved for extensions and future microarchitectures. Use `0`.

Description

The `KAPI_uniqueName2iid()` function looks up the numeric representation of an instruction based on its unique name and possibly an offset value in the case of access to numbered application registers. The unique name is derived from the `inst_ids.h` enumerated type names.

The function returns the instruction ID for *pchName*.

KAPI Operand Functions

These functions allow you to work with KAPI operand types.

The following enumeration defines operand roles for use with the function `KAPI_GetOppIndex()`.

```
typedef enum
{
    kapi_op_role_dest,
    kapi_op_role_src
} kapi_operand_role_e;
```

KAPI_GetOppIndex()

Gets operand index for instruction operand

```
extern int KAPI_GetOppIndex(void *pConfig, kapi_iid_t iid, int iIndex,
kapi_operand_role_e iRole, char *pchOppName);
```

Parameters

pConfig	The pointer intialized by `KAPI_init()`.
iid	Instruction ID.
iIndex	The operand index.
iRole	The role of the operand, whether source or destination.
pchOppName	Operand name.

Description

The function `KAPI_GetOppIndex()` uses the operand index according to the instruction description in the *Intel® Itanium™ Architecture Software Developer's Manual* to determine the KAPI operand index.

KAPI_oppGetSource( ), KAPI_oppGetDest( )

Gets operand index for functional unit operand name

```
int KAPI_oppGetSource( void *pConfig, kapi_fu_t fu, char *pchOppName )
int KAPI_oppGetDest( void *pConfig, kapi_fu_t fu, char *pchOppName )
```

Parameters

pConfig	The pointer intialized by `KAPI_init()`.
fu	Functional unit class.
pchOppName	Operand name.

Description

Given the name of a source or destination register and a functional unit class, the `KAPI_oppGetSource()` and `KAPI_oppGetDest()` queries return the integer representation of that operand needed for other KAPI functions. To get the primary operand number, pass a `NULL` pointer.

KAPI_srcOppCount( ), KAPI_destOppCount( )

Returns the number of source or destination operands for a functional unit class

```
int KAPI_srcOppCount( void *pConfig, kapi_fu_t fuSrc )
int KAPI_destOppCount( void *pConfig, kapi_fu_t fuDest )
```

Parameters

pConfig	The pointer intialized by `KAPI_init()`.
fuSrc	Source operand.
fuDest	Destination operand.

Description

The `KAPI_srcOppCount()` and `KAPI_destOppCount()` queries return the number of source or destination operands that have different latency/bypass characteristics for the given `kapi_fu_t`. Almost all instructions return `1` for both functions.

KAPI_srcOppName( ), KAPI_destOppName( )

Returns the string representation of the name of the source or destination

```
char *KAPI_srcOppName( void *pConfig, kapi_fu_t fuSrc, int opp );
char *KAPI_destOppName( void *pConfig, kapi_fu_t fuDest, int opp );
```

Parameters

pConfig	The pointer intialized by `KAPI_init()`.
fuSrc	Source functional unit.
opp	Operand index.
fuDest	Destination functional unit.

Description

The `KAPI_srcOppName()` and `KAPI_destOppName()` queries return the string representation of the name of the source or destination operands for this `fu_t`.

Ports and Clustering Functions

These functions deal with ports and functional unit classes.

KAPI_BundleIssueWidth( )

Returns the maximum number of bundles

```
int KAPI_BundleIssueWidth( void *pConfig, kapi_cluster_t cluster )
```

Parameters

pConfig	The pointer intialized by `KAPI_init()`.
cluster	Cluster index.

Description

The `KAPI_BundleIssueWidth()` query returns the maximum number of bundles that can be issued per cycle for a given cluster.

KAPI_DisperseCount4syl( )

Returns the number of instruction slots issued in a single cycle

```
int KAPI_DisperseCount4syl( void *pConfig, kapi_syl_t sylType )
```

Parameters

pConfig	The pointer intialized by `KAPI_init()`.
sylType	Instruction slot type. See Chapter 5, "Accessing Intel® Itanium™ Architecture Information".

Description

The `KAPI_DisperseCount4syl()` query returns the total number of instruction slots of the specified type that can be issued in a single cycle.

KAPI_fuCount( )

Returns the number of functional unit classes defined

```
int KAPI_fuCount( void *pConfig )
```

Parameters

pConfig	The pointer intialized by `KAPI_init()`.

Description

The `KAPI_fuCount()` query returns the number of functional unit classes defined.

KAPI_clusterCount( )

Returns the number of clusters defined

```
int KAPI_clusterCount( void *pConfig )
```

Parameters

pConfig	The pointer intialized by KAPI_init().

Description

The KAPI_clusterCount() query returns the number of clusters defined.

KAPI_cportMask4ut( )

Returns a 32-bit bitmask indicating which ports correspond to the indicated unit type

```
bv32_t KAPI_cportMask4ut( void *pConfig, kapi_cluster_t cluster,
kapi_ut_t ut)
```

Parameters

pConfig	The pointer intialized by KAPI_init().
cluster	Cluster index.
ut	Unit type.

Description

The KAPI_cportMask4ut() query returns a 32-bit bitmask indicating which ports are of the indicated unit type (*ut)* for a given cluster. This definition implies that at most 32 units are supported by the interface.

See the description in "KAPI_cportMask4fu( )".

KAPI_cportMask4fu( )

Indicating which cports can execute instructions of a specific fu class

```
bv32_t KAPI_cportMask4fu( void *pConfig, kapi_cluster_t cluster,
kapi_fu_t fu )
```

Parameters

pConfig	The pointer intialized by `KAPI_init()`.
cluster	Cluster index.
fu	Functional unit class.

Description

The `KAPI_cportMask4fu()` query returns a 32-bit bitmask indicating which cports can execute instructions with this functional unit class on the specified cluster. This definition implies that at most 32 units exist in the API. Wildcard (`cluster == -1`) is not supported.

The 32-bit vector received has a bit set for each cport on which instructions from this functional unit class can execute. Bits correspond to the `ut_t` order, and the bits for each *ut* can be found with the function `KAPI_cportMask4ut()`, or correlation between `KAPI_cportCount4ut()` and `ut_t` order.

KAPI_cportCount( )

Returns the number of cports available in the specified cluster

```
int KAPI_cportCount( void *pConfig, kapi_cluster_t cluster )
```

Parameters

pConfig	The pointer intialized by KAPI_init().
cluster	Cluster index.

Description

The KAPI_cportCount() query returns the number of cports available in the specified cluster. If *cluster* is -1, it returns the total number of global ports (all clusters).

KAPI_cportCount4ut( )

Returns the number of units of the specified type in the specified cluster

```
int KAPI_cportCount4ut( void *pConfig, kapi_cluster_t cluster,
kapi_ut_t ut )
```

Parameters

pConfig	The pointer intialized by KAPI_init().
cluster	Cluster index.
ut	Unit type.

Description

The KAPI_cportCount4ut() query returns the number of units of the specified type in the specified cluster. If *cluster* is -1, it returns the number of matching cports for all clusters.

KAPI_cportCount4fu( )

Returns the number of cports that can execute instructions of that type

```
int KAPI_cportCount4fu( void *pConfig, kapi_cluster_t cluster,
kapi_fu_t fu )
```

Parameters

fu	Functional unit class.
cluster	Cluster index.
pConfig	The pointer intialized by `KAPI_init()`.

Description

Given a functional unit class *fu*, the `KAPI_cportCount4fu()` query returns the number of cports that can execute instructions of that type for the specified cluster. This does not imply that all such functional units are identical or that many instructions can be dispersed in a given cycle. If *cluster* is `-1`, it returns the count of functional units for all clusters.

KAPI_portInfo( )

Gets info for machine-wide port

```
void KAPI_portInfo( void *pConfig, kapi_port_t port,
kapi_cluster_t *pcluster, kapi_cport_t *pcport, kapi_ut_t *put,
kapi_cutport_t *pcutport )
```

Parameters

pConfig	The pointer intialized by `KAPI_init()`.
port	Information is retrieved for this port.
pcluster	Cluster index.
pcport	Cport index.
put	Unit type index.
pcutport	Cutport index.

Description

The `KAPI_portInfo()` function maps a machine-wide port index to a cluster/cport pair and a cluster/functional unit/cutport triplet.

The *port* parameter must have a value between `0` and the number of ports minus one, otherwise the results of this function are undefined and could result in program failure.

KAPI_cportInfo( )

Gets info for port in specific cluster

```
void KAPI_cportInfo( void *pConfig, kapi_cluster_t cluster,
kapi_cport_t cport, kapi_port_t *pport, kapi_ut_t *put,
kapi_cutport_t *pcutport )
```

Parameters

pConfig	The pointer intialized by KAPI_init().
cluster	Cluster index.
cport	Cport index.
pport	Port index.
put	Unit type index.
pcutport	Cutport index.

Description

The KAPI_cportInfo() function maps a cluster/cport pair to a machine wide port index and a cluster/functional unit/cutport triplet.

The *cluster* and *cport* parameters must have a value representing a valid port/cluster pair (-1 is not allowed), otherwise the results of this function are undefined and could result in program failure.

KAPI_cutportInfo( )

Gets info for port in specific cluster and functional unit

```
void KAPI_cutportInfo( void *pConfig, kapi_cluster_t cluster,
kapi_ut_t ut, kapi_cutport_t cutport, kapi_port_t *pport,
kapi_cport_t *pcport )
```

Parameters

pConfig	The pointer intialized by KAPI_init().
cluster	Cluster index.
ut	Unit type.
cutport	Cutport index.
pport	Port index.
pcport	Cport index.

Description

The KAPI_cutportInfo() function maps cluster/functional unit/cutport triplet to a machine wide port index and a cluster/cport pair.

The cluster parameter must have a value between 0 and the number of clusters minus one, otherwise the results of this function are undefined and could result in program failure.

The *ut* and *cutport* parameters must both be -1 or neither be -1. If both *ut* and *cutport* parameters are -1, then the *cport* and *port* parameters are set to -1 upon return.

Instruction Latency Functions

All the functions dealing with latency are described in this section.

NOTE. *KAPI's ability to represent latency is limited by the API, the format of the knobsfile, and the complexity of various Itanium architecture implementations. Thus, the latency information and its expression in KAPI cannot fully represent the behavior of real processors in all cases.*

Computing Latency

The total latency of a pair of instructions is computed by adding several components together:

The core latency is given by the `KAPI_CoreLatency()` function. This function returns the basic latency of the execution, not including any time required to bypass the result to other functional units. The core latency is normally a function of the functional unit class and the destination operand (sometimes the source operand is used depending on the value of the `operand_direction` array), and is derived from the `fuLatency` variable and the `LATENCY` attribute in the KAPI file.

If the producer and consumer instructions are on the same cluster, the intracluster bypass latency plus the core latency equals the total latency for the specified instructions. The `KAPI_IntraClusterBypass()` function returns the intracluster bypass latency. The intracluster bypass latency is derived from the `INTRACLUSTER` attribute in the KAPI file.

If the producer and consumer instructions are on different clusters, the intercluster bypass plus the intercluster distance plus the core latency equals the total latency. The intercluster distance is a set amount that is added to all latencies between two clusters. This allows the distance between clusters to be easily lengthened or shortened without re-specifying all the bypass entries. The intercluster bypass latency is retrieved using the `KAPI_InterClusterBypass()` function whose values are derived from the `INTERCLUSTER` attribute.

The intercluster separation is retrieved from the `KAPI_ClusterDistance()` function that is based on the `CLUSTERDISTANCE` KAPI attribute.

The function `KAPI_TotalLatency()` enables convenient acces to the total latency (core + bypasses) between two functional units.

Accuracy of Latency Information

The following enumerations are used with the functions `KAPI_fuGetMiscInfo()` and `KAPI_fuGetLatencyType()`, respectively.

```
typedef enum
{
    kapi_fu_info_approximate_latency,
    kapi_fu_info_no_latency_info
} kapi_fu_infobits_e;
```

The function `KAPI_fuGetLatencyType()` uses the `kapi_latency_type_t` enumeration to indicate the accuracy of the latency information contained in the knobsfile for a specific functional unit class.

```
typedef enum
{
    kapi_latency_type_none,
    kapi_latency_type_approximate,
    kapi_latency_type_full
} kapi_latency_type_t;
```

KAPI_fuGetMiscInfo()

Gets information about a functional unit class

```
extern bv32_t KAPI_fuGetMiscInfo(void *pConfig, kapi_fu_t fu);
```

Parameters

pConfig	The pointer intialized by `KAPI_init()`.
fu	Functional unit class.

Description

The function `KAPI_fuGetMiscInfo()` retrieves a bit vector that signifies properties associated with the functional unit class.

KAPI_fuGetLatencyType()

Finds latency accuracy

```
extern kapi_latency_type_t KAPI_fuGetLatencyType(void *pConfig,
kapi_fu_t fu);
```

Parameters

pConfig	The pointer intialized by `KAPI_init()`.
fu	Functional unit class.

Description

The function `KAPI_fuGetLatencyType()` returns the accuracy of latency information contained in the knobsfile for a specified functional unit. Use it before calling any other latency-related function.

KAPI_CoreLatency( )

Returns the base execution latency for the given `fu_t`/`opp` *pair*

```
int KAPI_CoreLatency( void *pConfig, kapi_fu_t fuProd, int oppProd )
```

Parameters

pConfig	The pointer intialized by `KAPI_init()`.
fuProd	Functional unit class producing the operand.
oppProd	Index of operand (source or destination) in producing the functional unit.

Description

The `KAPI_CoreLatency()` function returns the base execution latency for the given `kapi_fu_t` and operand pair. If `0` is passed to *oppProd*, then the latency for the primary source or destination register is returned, depending on the value of `operand_direction[ fuProd ]`.

KAPI_InterClusterBypass( )

Returns the component of total latency due to inter-cluster delays

```
int KAPI_InterClusterBypass( void *pConfig,
kapi_cluster_t clusterProd, kapi_fu_t fuProd, int oppProd,
kapi_ut_t utProd, kapi_cutport_t cutportProd,
kapi_cluster_t clusterCons, kapi_fu_t fuCons, int oppCons,
kapi_ut_t utCons, kapi_cutport_t cutportCons )
```

Parameters

pConfig	The pointer intialized by `KAPI_init()`.
clusterProd	Cluster index in which the unit producing the operand resides.
fuProd	Functional unit class producing the operand.
oppProd	Index of the destination operand when producing the functional unit.
utProd	Index of the unit on which the producer operation executes.
cutportProd	Cutport index of the port to which the prducer operation maps.
clusterCons	Cluster index in which the unit consuming the operand resides.
fuCons	Functional unit class consuming the operand.
oppCons	Index of the source operand in consuming functional unit.
utCons	Index of the unit on which the consumer operation executes.
cutportCons	Cutport index of the port to which the consumer operation maps.

Description

The `KAPI_InterClusterBypass()` function returns the component of total latency due to inter-cluster delays as a function of clusters, functional unit classes, operands, and ports. There are no wildcard specifications.

If `clusterProd` and `clusterCons` are the same, the function returns `0`.

Ports in this function are specified by their cutport/ut number pair.

KAPI_IntraClusterBypass( )

Returns the number of cycles required to bypass the specified value

```
int KAPI_IntraClusterBypass( void *pConfig, kapi_cluster_t cluster,
kapi_fu_t fuProd, int oppProd, kapi_ut_t utProd,
kapi_cutport_t cutportProd, kapi_fu_t fuCons, int oppCons,
kapi_ut_t utCons, kapi_cutport_t cutportCons )
```

Parameters

pConfig	The pointer intialized by KAPI_init().
cluster	Cluster index in which the units reside.
fuProd	Functional unit class producing the operand.
oppProd	Index of the destination operand in producing functional unit.
utProd	Index of the unit on which the producer operation executes.
cutportProd	Cutport index of the port to which the prducer operation maps.
fuCons	Functional unit class consuming the operand.
oppCons	Index of the source operand in consuming functional unit.
utCons	Index of the unit on which the consumer operation executes.
cutportCons	Cutport index of the port to which the consumer operation maps.

Description

The KAPI_IntraClusterBypass() function returns the number of cycles required to bypass the specified value from source instruction to consumer instruction given a fully specified producer/consumer pair. The cluster must be specified as part of this function; no wildcard arguments are supported.

KAPI_ClusterDistance( )

Returns the number of cycles between two clusters

```
int KAPI_ClusterDistance( void *pConfig, kapi_cluster_t clusterProd,
kapi_cluster_t clusterCons )
```

Parameters

pConfig	The pointer intialized by `KAPI_init()`.
clusterProd	Cluster index in which the unit producing the operand resides.
clusterCons	Cluster index in which the unit consuming the operand resides.

Description

The `KAPI_ClusterDistance()` function returns the fixed number of cycles added to all results that flow from `clusterProd` to `clusterCons`.

KAPI_TotalLatency( )

Returns the total latency

```
int KAPI_TotalLatency( void *pConfig, kapi_cluster_t clusterProd,
kapi_fu_t fuProd int oppProd, kapi_ut_t utProd,
kapi_cutport_t cutportProd, kapi_cluster_t clusterCons,
kapi_fu_t fuCons, int oppCons, kapi_ut_t utCons,
kapi_cutport_t cutportCons )
```

Parameters

pConfig	The pointer intialized by `KAPI_init()`.
clusterProd	Cluster index in which the unit producing the operand resides.
fuProd	Functional unit class producing the operand.
oppProd	Index of the destination operand in producing functional unit.
utProd	Index of the unit on which the producer operation executes.
cutportProd	Cutport index of the port to which the prducer operation maps.
clusterCons	Cluster index in which the unit consuming the operand resides.
fuCons	Functional unit class consuming the operand.
oppCons	Index of the source operand in consuming functional unit.
utCons	Index of the unit on which the consumer operation executes.
cutportCons	Cutport index of the port to which the consumer operation maps.

Description

The `KAPI_TotalLatency()` function returns the total latency. Restrictions on wildcards and special cases are as restrictive as the most restrictive of the subcalls below.

If this is `INTERcluster`,

```
result = KAPI_InterClusterBypass( pConfig, clusterProd, fuProd,
    utProd, cutportProd, oppProd, clusterCons, fuCons, utCons,
    cutportCons, oppCons )
```

```
    + KAPI_CoreLatency( pConfig, fuProd, oppProd )
    + KAPI_ClusterDistance( pConfig, clusterProd, clusterCons )
```

If this is `INTRAcluster`,

```
result = KAPI_IntraClusterBypass( pConfig, cluster, fuProd, oppProd,
    utProd, cutportProd, fuCons, oppCons, cportCons )
      + KAPI_CoreLatency( pConfig, fuProd, oppProd )
```

KAPI_IntraClusterBypassList( )

Returns list of intracluster bypasses for a cutport

```
papair_t *KAPI_IntraClusterBypassList( void *pConfig,
kapi_cluster_t clr, kapi_fu_t fuProd, int oppProd, kapi_ut_t utProd,
kapi_cutport_t cutportProd, kapi_fu_t fuCons, int oppCons,
kapi_ut_t utCons, kapi_cutport_t cutportCons, int *pnbypass )
```

Parameters

pConfig	The pointer intialized by `KAPI_init()`.
clr	Cluster index in which the units reside.
fuProd	Functional unit class producing the operand.
oppProd	Index of the destination operand in producing functional unit.
utProd	Index of the unit on which the producer operation executes.
cutportProd	Cutport index of the port to which the producer operation maps.
fuCons	Functional unit class consuming the operand.
oppCons	Index of the source operand in consuming functional unit.
utCons	Index of the unit on which the consumer operation executes.
cutportCons	Cutport index of the port to which the consumer operation maps.
pnbypass	The number of bypasses.

Description

The KAPI_IntraClusterBypassList() function returns a pointer to an array of matching active intracluster bypass entries from the knobsfile. The number of matching entries is returned through the pnbypass argument. Any of the input arguments can be specified as -1 in which case wildcard searches are performed. papair_t is defined in kapi_ia64.h. The array pointed to is statically allocated and is only valid until the next function call to KAPI_IntraclusterBypassList(). The user must make any desired copies.

```
typedef struct PAPAIR_T {
    kapi_cluster_t          cluster;
    kapi_fu_t               fuSrc, fuTar;
    kapi_cutport_t          cutportSrc, cutportTar;
    kapi_ut_t               utSrc, utTar;
    int                     oppSrc, oppTar;
    int                     iValue;
} papair_t;
```

where

cluster	Source cluster.
fuSrc, fuTar	Source and destination functional unit class.
cutportSrc, cutportTar	Source and destination port index for unit.
utSrc, utTar	Source and destination unit.
oppSrc, oppTar	Source and destination operand index.
iValue	Bypass value.

The client is responsible for deallocating the memory for the structure returned by this function.

KAPI_InterClusterBypassList( )

Returns a pointer to an array of matching active inter cluster bypass entries from the knobsfile

```
pepair_t *KAPI_InterClusterBypassList( void *pConfig,
kapi_cluster_t clusterProd, kapi_fu_t fuProd, int oppProd,
kapi_ut_t utProd, kapi_cutport_t cutportProd,
kapi_cluster_t clusterCons,kapi_fu_t fuCons, int oppCons,
kapi_ut_t utCons, kapi_cutport_t cutportCons, int *pnbypass)
```

Parameters

pConfig	The pointer intialized by KAPI_init().
clusterProd	Cluster index in which the unit producing the operand resides.
fuProd	Functional unit class producing the operand.
oppProd	Index of the destination operand in producing functional unit.
utProd	Index of the unit on which the producer operation executes.
cutportProd	Cutport index of the port to which the prducer operation maps.
clusterCons	Cluster index in which the unit consuming the operand resides.
fuCons	Functional unit class consuming the operand.
oppCons	Index of the source operand in consuming functional unit.
utCons	Index of the unit on which the consumer operation executes.
cutportCons	Cutport index of the port to which the consumer operation maps.
pnbypass	The number of bypasses.

Description

The KAPI_InterClusterBypassList() function returns a pointer to an array of matching active inter cluster bypass entries from the knobsfile. The number of matching entries is returned through the pnbypass argument. Any of the input arguments can be specified as -1, in which case wildcard searches are performed. pepair_t is defined in

kapi_ia64.h. The array returned by this function is statically allocated and only exists until the next function call to KAPI_InterclusterBypassList(). The user must make any desired copies.

```
typedef struct PEPAIR_T {
    kapi_cluster_tclusterSrc, clusterTar;
    kapi_fu_t        fuSrc, fuTar;
    kapi_cutport_t  cutportSrc, cutportTar;
    kapi_ut_t       utSrc, utTar;
    int              oppSrc, oppTar;
    int              iValue;
} pepair_t;
```

where

clusterSrc, clusterTar	Source and destination cluster.
fuSrc, fuTar	Source and destination functional unit class.
cutportSrc, cutportTar	Source and destination port index for unit.
utSrc, utTar	Source and destination unit.
oppSrc, oppTar	Source and destination operand index.
iValue	Bypass value.

The client is responsible for deallocating the memory for the structure returned by this function.

KAPI_MinIntraClusterTotalLatency( )

Returns the minimum latency between instructions

```
int KAPI_MinIntraClusterTotalLatency( void *pConfig,
kapi_cluster_t cluster, kapi_fu_t fuProd, int oppProd,
kapi_fu_t fuCons, int oppCons )
```

Parameters

pConfig	The pointer intialized by `KAPI_init()`.
cluster	Cluster index.
fuProd	Functional unit class producing the operand.
oppProd	Index of the destination operand in producing functional unit.
fuCons	Functional unit class consuming the operand.
oppCons	Index of the source operand in consuming functional unit.

Description

The `KAPI_MinIntraClusterTotalLatency()` function returns the minimum latency between instructions of the given types within the specified cluster.

Itanium Architecture Bundle Content Query Functions

The functions below provide information on instruction slots, bundle issue, and restrictions.

KAPI_SylCount_bid( )

Returns the number of instruction slots of type

```
int KAPI_SylCount_bid( void *pConfig, kapi_bid_t bid, kapi_syl_t syl )
```

Parameters

pConfig	The pointer intialized by `KAPI_init()`.
bid	Bundle ID index.
syl	Instruction slot index.

Description

The `KAPI_SylCount_bid()` function returns the number of instruction slots of the specified type in the given *bid*.

KAPI_SbitPlacement_bid( )

Checks for stop

```
int KAPI_SbitPlacement_bid( void *pConfig, kapi_bid_t bid )
```

Parameters

pConfig	The pointer intialized by `KAPI_init()`.
bid	Bundle ID index.

Description

The `KAPI_SbitPlacement_bid()` function returns `0` if there is no stop in the specified bundle ID. It returns `1` if the bundle has a stop between the first and second instruction slots, and `2` if there is a stop between the second and third instruction slots.

KAPI_isReserved_bid( )

Checks if bid is reserved

```
int KAPI_isReserved_bid( void *pConfig, kapi_bid_t bid )
```

Parameters

pConfig	The pointer intialized by `KAPI_init()`.
bid	Bundle ID index.

Description

The `KAPI_isReserved_bid()` function returns `0` if *bid* is architecturally reserved, otherwise it returns `1`.

KAPI_SylOrder_bid( )

Gets instruction slot type for each slot in the bundle

```
void KAPI_SylOrder_bid( void *pConfig, kapi_bid_t bid,
kapi_syl_t mpsyl[ 3 ] )
```

Parameters

pConfig	The pointer intialized by `KAPI_init()`.
bid	Bundle ID index.
mpsyl	Array containing bundle-width instruction slots.

Description

The `KAPI_SylOrder_bid()` function fills in the instruction slot values in the array *mpsyl* passed for the given *bid*.

KAPI_utCount_syl( )

Returns number of units needed for a instruction slot

```
void KAPI_utCount_syl(void *pConfig, kapi_syl_t syl, kapi_ut_t mput[])
```

Parameters

pConfig	The pointer intialized by `KAPI_init()`.
syl	Instruction slot index.
mput	Array containing the units needed for each instruction slot type.

Description

The `KAPI_utCount_syl()` function returns the number of units of each type required to execute a instruction slot of the specified type. This function is currently only really important for instruction slots. The length of `mput` must be `KAPI_utCount()`.

Cache Hierarchy Functions

The functions outlined below enable easier access to cache parameters. The `cache_t` structure holds all values pertinent to the cache.

```
typedef struct KAPI_CACHE_T {
    int nLines;         // number of lines in this cache
    int nBytesLine;     // number of bytes/line
    int nWays;          // associativity
    int nCyclesRead;    // load to use latency if reference hits
                        // in this cache
    int nCachePorts;    // number of access ports
                        // array indexed by nCachePorts
    kapi_cacheport_t cacheportInfo[KAPI_MAX_CACHE_PORTS_IMPL];
    bv32_t bv32CacheContents; // kind of cache: instruction or data
    kapi_cache_policy_write_e iWritePolicy; // write policy
    kapi_cache_policy_repl_e  iReplPolicy;  // replacement policy
    kapi_cache_policy_alloc_e iAllocPolicy; // allocation policy
} kapi_cache_t;
```

Each port is assumed to be able to load contiguous data only, and only one request (i.e., data from only one load or one store) can be met. Thus, this information does not accurately represent bus widths between levels of the cache, rather, only the number of independent requests that can be serviced simultaneously.

Ports for which both the read (R) and write (W) bit are set are shared R/W ports and only one R or one W per cycle can be accommodated.

There is room for expansion of this structure.

```
struct KAPI_CACHEPORT_T {
    bv32_t bv32accessmode;
} kapi_cacheport_t;
```

KAPI_CacheHierarchy()

Gets cache hierarchy information

```
kapi_cache_t * KAPI_CacheHierarchy( void *pConfig, int level,
kapi_cache_types_e cachecontent );
```

Parameters

pConfig	The pointer intialized by KAPI_init().
level	Cache level.
cachecontent	Cache content type: instruction/data.

Description

The KAPI_CacheHierarchy() function returns a structure, which the user is responsible for deallocating, that contains information about the cache hierarchy as indicated in the structure comments.

This interface assumes that there are N levels (numbered 0 to N-1) of non-unified Icache, M levels of non-unified Dcache (numbered 0 to M-1) and Q levels of unified cache (numbered 0 to Q-1) which are parents of level N-1 of Icache and level M-1 of Dcache.

A Icache level N query therefore returns the values for the unified cache level 0.

Unified caches have bits set for both I and D contents. I-only and D-only only set their corresponding bit set.

This function returns NULL for illegal *cachecontent* (as given by KAPI_CACHECONTENT_xxxx enumerators defined in the header) values. It returns NULL for a *level* value higher than the number of levels in that type of hierarchy.

KAPI_nCacheHierarchyLevels()

Gets number of cache hierarchy levels

```
int* KAPI_nCacheHierarchyLevels ( void *pConfig,
kapi_cache_types_e cachecontent );
```

Parameters

pConfig	The pointer intialized by KAPI_init().
cachecontent	Cache content type: instruction/data.

Description

The KAPI_nCacheHierarchyLevels() function returns the number of levels in the cache hierarchy of the given type. This function does not take a bitmask, but only one of the valid *cachecontent* enum values (instruction/data).

The function returns 1 for illegal *cachecontent* values.

Saving Information in Header Format Functions

These functions are all meant to enable functions to the Itanium architecture layer of KAPI during run time, without saving the information in raw readable format, and to save the time spent on initializing all of these structures.

When enabling Itanium architecture functions from header, rather then using KAPI to parse a knobs file, the data structures can only accept the structures initialized at build time. No functionality is provided for a "delta" structure, unlike the ability to have a "delta" knobs file. In essence, these functions provide a shortcut to users who need to use only the Itanium architecture layer functions of KAPI during run time. Instead of building user-defined structures during build time, and code to understand these structures, or parsing during runtime, the client can use these functions to save the structures needed by the Itanium architecture aware layer during build time.

An added benefit is that these structures are only human-readable in the source code, and in the executable file they are saved in binary format.

Using this set of functions is a 2-step process:

Step 1 Use the relevant Save function during build time to build a source file to be compiled and linked with your project.

Step 2 Use the relevant Enable function in runtime, and then use the appropriate function from the Itanium architecture layer.

NOTE. *When using this model, no strings are saved in the header for security reasons, and KAPI functions returning strings will not be available (most notably the functions returning EMdb instruction name and mnemonic).*

Sample usage:

During build, add a small program linked to KAPI that functions as follows:

```
KAPI_save_as_header_all_IA64_info
    (pknobs,kapi_file_header,kapi_info_ptr);
```

Include that file in your project, link it with the KAPI library, and then add:

```
KAPI_ fEnableIA64call_from_info_ptr_all_IA64_info(
    pknobs,kapi_info_ptr);
KAPI_cportCount( pknobs, -1 );
```

KAPI_save_as_header Functions

Save header information functions

The following functions share the same behavior and parameters:

```
KAPI_save_as_header_all_IA64_inf()
KAPI_save_as_header_latency_all_info()
KAPI_save_as_header_latency_core_info()
KAPI_save_as_header_cluster_all_info()
KAPI_save_as_header_cluster_distance_info()
KAPI_save_as_header_cluster_intracluster_latency_info()
KAPI_save_as_header_cluster_intercluster_latency_info()
KAPI_save_as_header_cluster_width_info()
KAPI_save_as_header_functional_units_info_info()
KAPI_save_as_header_cport_info()
KAPI_save_as_header_instruction_all_info()
KAPI_save_as_header_instruction_type_info()
KAPI_save_as_header_byd_n_syl_info()
```

For the above functions, the syntax is as follows:

```
Int <name>( FILE *fp, void *pConfig , char *pchName)
```

where:

pConfig	the knobs pointer
fp	the file pointer
pchName	the name to assign to the resulting structure

These functions save the relevant information structures in the indicated file as C structures. They return `0` when successful, and other values when unsuccessful.

KAPI_fEnableIA64call_from_header()

Point to a saved structure

```
int KAPI_fEnableIA64call_from_header(void **pConfig,
    void *pHeaderConfig, int iReserved);
```

where:

pHeaderConfig	pointer to the saved header structure
pConfig	the knobs pointer
iReserved	a reserved integer variable

The `KAPI_fEnableIA64call_from_header()` function points to the structure that was saved with one of the save header functions. It assigns the information pointer to *pConfig*. It returns `0` when successful, and other values when unsuccessful.

KMAPI: For Deriving Machine-State Information

8

The KMAPI layer provides better abstraction for the Intel® Itanium™ microarchitecture. This offshoot of KAPI implements generic functions needed by several tools whose output is too complex to express in table format. This is the third layer. These functions use information provided in the knobsfile addendum, and therefore require special initialization as defined in "Knobs/Functions Relationship". This chapter describes the usage model of the library and the set of functions (API) encapsulating various machine-state issues contained in the library.

The KMAPI library provides the following:

- reduces replication by defining a set of functions that implement common machine state issues needed by several tools
- reduces the effort required in switching tools to different implementations of the microarchitecture

Enumerations and Data Structures

This section describes the KMAPI semantics, which define parameters for use in functions.

Resource Status Enumeration

As executable instructions need to be allocated to a specific unit type and the number of such units is finite, the resource structures enable reference to mapping of instructions to units.

```
typedef enum KAPI_RESOURCE_STATUS_E {
    kmapi_resource_free=0,
    kmapi_resource_occupied=1,
```

```
    kmapi_resource_invalid
} kmapi_resource_status_e;
```

Resource Map Definitions

These type definitions indicate the available ports in the machine.

```
typedef int kmapi_resource_t;
typedef bv32_t *kmapi_pResourceMap_t;
typedef bv32_t kmapi_ResourceMap_t;
```

A resource map is an array that holds the resource status for each machine resource.

```
typedef struct {
    int nResources;
    kmapi_ResourceMap_t *dmResourceMaps;
} kmapi_Resource_Array_t;
```

The structure `kmapi_Resource_Array_t` holds an array of resource maps. KMAPI uses these resource maps internally to determine allocation. They can be retrieved using the function `KMAPI_GetRawOptions()`.

Instruction Bundle Information Structure

The machine resource definition may change for each implementation of the microarchitecture.

An issue group is composed of a number of bundles that are executed in the one cycle. For Merced, the number of bundles is two. KMAPI receives bundle information for each bundle in the issue group. This structure describes the place of the instruction in the issue group.

```
typedef struct _KMAPI_BUNDLE_INFO_T {
    int iPlace;        // place of bundle in issue group
    kapi_bid_t bid;    // bundle type
    int iSlot;         // instruction slot in the bundle
} kmapi_bundle_info_t;
```

Return Values Enumeration

KMAPI functions return this value to indicate success or failure.

```
typedef enum _KMAPI_RESULT_E {
    kmapi_success=0,
    kmapi_failure,
    kmapi_invalid_input,
    kmapi_internal_error
} kmapi_result;
```

Implicit Stop Enumeration

Stops can be implicitly inserted. The following enumeration lists the possibilities for implicit stops due to a specific bundle, such as the bundle `mmf` in Merced:

```
typedef enum _KMAPI_IMPLICIT_BREAK_E {
    kmapi_no_implicit_break,
    kmapi_implicit_break_before,
    kmapi_implicit_break_after,
    kmapi_implicit_break_both
} kmapi_implicit_break_t;
```

Allocation Option Structure

This structure maps allocation options.

```
typedef struct {
    int nOptions;             // size of following array
    bv32_t *dmpOptions;       // dynamically allocated array of options
    kapi_cluster_t *dmpClrID;//dynamically allocated array of clusters
} kmapi_allocation_option_t;
```

Using KMAPI

KMAPI use relies on the knobs defined in the addendum knobsfile and therefore requires the KAPI library for initialization. Accordingly, the functions outlined in this section are divided into these distinct groups:

- initialization functions
- mapping functions
- machine state-related functions

Initialization Functions

These functions initialize KMAPI. They are needed in addition to the `KAPI_ia64_Initialize()` function.

KMAPI_initialize()

Initializes KMAPI

```
void *KMAPI_initialize(void *pIA64Config);
```

Parameters

`pIA64Config` Itanium architecture KAPI pointer to an information structure.

Description

To initialize KMAPI, use a KAPI structure that was initialized previously to allow layer 2 (Itanium architecture) calls. Use the pointer returned by the function in further calls to KMAPI functions.

The function returns `NULL` if unsuccessful.

KMAPI_save_as_header_all_kmapi_info()

Saves KMAPI info

```
kmapi_result KMAPI_save_as_header_all_kmapi_info( FILE *fp,
void *pConfig, char *pchKmapiInfoName, char *pchKapiIA64InfoName);
```

Parameters

fp	File pointer.
pConfig	KMAPI information pointer, pointing to an initialized KMAPI information structure.
pchKmapiInfoName	Name assigned to the resulting structure.
pchKapiIA64InfoName	Name for the Itanium architecture information structure.

Description

The KMAPI_save_as_header_all_kmapi_info() function saves the KMAPI information initialized by KMAPI_initialize() to file *fp*, using the name *pchKmapiInfoName*. If you did not save the KAPI information that was used to initialize KMAPI separately, you can save it now using *pchKapiIA64InfoName*. Otherwise use NULL for that parameter, and enable KMAPI calls from header with the Itanium architecture configuration saved separately. Compile the resulting file to an object, add this object to your project, and use *pchKmapiInfoName* (and *pchKapiIA64InfoName* if the name was not NULL in the saver) with the function KMAPI_fEnableKmapiCalls_from_header().

The relevant information structures are saved in the indicated file as C structures.

KMAPI_fEnableKmapiCalls_from_header()

Enables KMAPI info

```
kmapi_result KMAPI_fEnableKmapiCalls_from_header (void **pConfig,
void *pKMAPI_HeaderConfig, void *pKAPI_IA64_HeaderConfig,
int iReserved);
```

Parameters

pConfig	A pointer to the configuration holder to send later as a parameter for KMAPI functions.
pKMAPI_HeaderConfig	The KMAPI knobs pointer.
pKAPI_IA64_HeaderConfig	The Itanium architecture knobs pointer.
iReserved	A reserved integer variable.

Description

The `KMAPI_fEnableKmapiCalls_from_header()` function assigns the information pointer to the appropriate point in the knobs structure.

Mapping-related Functions

This section describes functions that deal with the mapping of instructions to ports, according to ports available and optimal performance.

KMAPI_MapInstructionToPort()

Maps an instruction

```
kmapi_result KMAPI_MapInstructionToPort(void *pConfig,
kmapi_pResourceMap_t mpMap, kmapi_bundle_info_t *pbinfo,
kapi_iid_t iid, kmapi_pResourceMap_t pAllocatedResource);
```

Parameters

pConfig	The pointer intialized by KMAPI_init().
mpMap	Resource map of the current status.
pbinfo	Bundle information for instruction.
iid	Instruction ID.
pAllocatedResource	Address of map after allocation. If this is the same as *mpMap*, an update occurs. Otherwise, only the resources that were allocated are marked as occupied.

Description

The KMAPI_MapInstructionToPort() function maps an instruction according to rules and resources defined in the addendum knobsfile, as well as internal logic that is too complicated to put in a knobs file.

This function returns the following values:

kmapi_success	Allocation succeeded
kmapi_failure	Allocation failed
kmapi_invalid_input	*pConfig* contains information that KMAPI can't use for mapping, or a parameter contains illegal information.

`kmapi_internal_error` A fatal error occurred.

KMAPI_CreateResourceMap()

Creates a resource map

```
kmapi_pResourceMap_t KMAPI_CreateResourceMap(void *pConfig);
```

Parameters

pConfig The pointer intialized by `KMAPI_init()`.

Description

The `KMAPI_CreateResourceMap()` function returns a resource map pointer.

KMAPI_DestroyResourceMap()

Destroys a resource map

```
void KMAPI_DestroyResourceMap(kmapi_pResourceMap_t mpMap);
```

Parameters

mpMap Resource map of the current status.

Description

The `KMAPI_DestroyResourceMap()` function deallocates memory from a resource map.

KMAPI_ClearResourceMap()

Clears a resource map

```
void KMAPI_ClearResourceMap(kmapi_pResourceMap_t mpMap);
```

Parameters

mpMap	Resource map of the current status.

Description

The `KMAPI_ClearResourceMap()` function clears data from a resource map, without deallocating memory.

KMAPI_SetResource()

Sets the status

```
kmapi_result KMAPI_SetResource(kmapi_pResourceMap_t mpMap,
kmapi_resource_t iResource, kmapi_resource_status_e status);
```

Parameters

mpMap	Resource map of the current status.
iResource	Resource index.
status	Resource status, whether occupied or free. See "Enumerations and Data Structures".

Description

The `KMAPI_SetResource()` function sets a single resource in a map to a status.

KMAPI_SetResourceMap()

Copies the status

```
kmapi_result KMAPI_SetResourceMap(kmapi_pResourceMap_t mpMapDst,
kmapi_pResourceMap_t mpMapSrc, kmapi_resource_status_e status);
```

Parameters

mpMapDst	Resource map destination.
mpMapSrc	Resource map source.
status	Resource status, whether occupied or free. See "Enumerations and Data Structures".

Description

The KMAPI_SetResourceMap() function sets all the resources marked in *mpMapSrc* to status in *mpMapDst*.

KMAPI_Resource_Status()

Checks the status

```
kmapi_resource_status_t KMAPI_Resource_Status(kmapi_pResourceMap_t mpMap,
kmapi_resource_t iResource);
```

Parameters

mpMap	Resource map of the current status.
iResource	Resource index.

Description

The `KMAPI_Resource_Status()` function checks the status of a specific resource in a map.

KMAPI_GetFirstAllocatedMapResource(), KMAPI_GetNextAllocatedMapResource()

Gets the resources

```
kmapi_resource_t KMAPI_GetFirstAllocatedMapResource(bv32_t *pbvMap);
kmapi_resource_t KMAPI_GetNextAllocatedMapResource(bv32_t *pbvMap,
kmapi_resource_t iPrevResource);
```

Parameters

pbvMap	Pointer to the map holding the information.
iPrevResource	A resource index received from a previous call to one of the functions.

Description

The `KMAPI_GetFirstAllocatedMapResource()` and `MAPI_GetNextAllocatedMapResource()` functions extract all the resources from a map, one at a time.

KMAPI_resource_info()

Gets KAPI info

```
kmapi_resultKMAPI_resource_info(void*pConfig,kmapi_resource_tiResource,
kapi_port_t *pport, kapi_cluster_t *pclr, kapi_cport_t *pcport,
kapi_ut_t *put, kapi_cutport_t *pcutport);
```

Parameters

pConfig	The pointer intialized by `KMAPI_init()`.
iResource	Resource index.
pport	Port index.
pclr	Cluster index.
pcport	Cport index.
put	Unit type index.
pcutport	Cutport index.

Description

The `KMAPI_resource_info()` function gets KAPI information for a specific KMAPI resource.

KMAPI_AllocationOptions4fu()

Gets mapping options

```
kmapi_result KMAPI_AllocationOptions4fu(void *pConfig, kapi_fu_t fu,
kmapi_allocation_option_t *pOptions);
```

Parameters

pConfig	The pointer intialized by KMAPI_init().
fu	Functional unit class.
pOptions	A list of possible mapping options. See "Allocation Option Structure".

Description

The KMAPI_AllocationOptions4fu() function gets a list of possible mapping options for a specific *fu* class.

KMAPI_ClearAllocationOptions()

Deallocates memory

```
kmapi_result KMAPI_ClearAllocationOptions(
kmapi_allocation_option_t *pOptions)
```

Parameters

pOptions	A list of possible mapping options. See "Allocation Option Structure".

Description

The KMAPI_ClearAllocationOptions() function deallocates memory used by the allocation option structure.

KMAPI_nAllocationOptions4syl()

Get number of mapping options

```
kmapi_result KMAPI_nAllocationOptions4syl(void *pConfig,
kapi_syl_t isyl, int *iOptions);
```

Parameters

pConfig	The pointer intialized by `KMAPI_init()`.
iSyl	Instruction slot type.
iOptions	Number of options returned by the function.

Description

The function `KMAPI_nAllocationOptions4syl()` returns the total number of allocations possible for a specific instruction slot type.

For example, it returns 2 for an I instruction slot on Itanium™ processors.

KMAPI_GetRawOptions()

Gets allocation option data

```
kmapi_Resource_Array_t *KMAPI_GetRawOptions(void *pConfig,
int iIssueSlot, kapi_syl_t iSyl);
```

Parameters

pConfig	The pointer intialized by KMAPI_init().
iIssueSlot	Slot number within the issue group. Valid values: 0..Max issue per cycle.
iSyl	Instruction slot type

Description

The function KMAPI_GetRawOptions() retrieves raw allocation option data from the slot and instruction slot type.

Deallocation for the data is the client's responsibility.

Split Issue-Related Functions

This section describes the functions that deal with the positioning of splits between sets of instructions.

KAPI_ImplicitBreakType()

Gets an implicit break

```
kmapi_implicit_break_t  KAPI_ImplicitBreakType(void *pConfig,
kapi_bid_t bid, kapi_fu_t mpfu[3]);
```

Parameters

pConfig	The pointer intialized by `KMAPI_init()`.
bid	Bundle type.
mpfu	fuClasses of the instructions in the bundle.

Description

The `KAPI_ImplicitBreakType()` function reports any implicit stops, whether before or after the bundle, depending on the bundle type and functional unit classes used.

KAPI_ImplicitBreakTypeForInst()

Gets an implicit break for instructions

```
kmapi_implicit_break_t  KAPI_ImplicitBreakTypeForInst(void *pConfig,
kapi_bid_t bid, kapi_iid_t mpid[3]);
```

Parameters

pConfig	The pointer intialized by KMAPI_init().
bid	Bundle type.
mpid	KAPI instruction ID of the instructions in the bundle.

Description

The KAPI_ImplicitBreakTypeForInst() function reports any implicit stops, whether before or after the bundle, depending on the bundle type and instructions used.

Knobs/Functions Relationship

This section describes the relationships between knobs and functions. To function properly, all the machine state-related functions require definition of the following knobs:

- dispersal option, for mapping
- dispersal exception, for mapping
- split issue rules
- UARCH

These knobs are described in greater detail in the following sections.

Mapping

For mapping purposes, KMAPI expects the knobsfile defining dispersal options and dispersal exceptions.

The mapping formats are instruction slot-oriented.

Dispersal Option

KMAPI relies on the rules of dispersal option for finding mapping options.

Syntax

```
(xxx,xxx)=(r1+r2,r3....)
```

Parameters

x	is one of the following instruction slots:	
	M	memory
	I	immediate
	F	floating-point
	B	branch
	L	long
r1, r2, ...	are the available resources (for Merced that means: `clusterX1\utT/cportT1+clusterZ1\utT/cportT2,clusterX2\utT/cportT3` etc....)	

Example

```
DISPERSAL_OPTION += "(M--,---)=(cluster0\utM/cportM0)";
```

Dispersal Exception

KMAPI relies on the rules of dispersal exception for finding mapping options.

Syntax

```
(b1,b2[..])/(yyy,yyy[..])/(f1,f2...)=(r1+r2,r3...)
```

Parameters

b1	is the first template, *b2* is the second template, "ANY" for any
one *y*	is the instruction slot affected by this exeption
the other *y*	must all be -
f1, *f2*, . . .	are the fu classes for which the exeption is valid, or "ANY" for any

r1, *r2*, . . .	are the available resources for this specific set of conditions, overriding the dispersal option

Example

```
DISPERSAL_EXCEPTION +=
"(bidMIB,ANY)/(--B,---)/(fuNOP,fuBRP)=(cluster0\utB/cportB0)";
```

Split Issue

For split issue purposes, KMAPI expects knobs that provide the information for split issue rules:

The format is bundle-oriented, therefore there are always three in the group "`-/-/-`".

Syntax

"*bid*:*Y1*:*Y2*:fu(-/-/-,-/-/-,…)"

Parameters

bid	is a bundle ID, as defined by `bid_t`
Y1	`yes` or `no` signify split issue before bundle execution
Y2	`yes` or `no` signify split issue after bundle execution
-	is any of the `fuClass` classes defined in `fu_t`, or `*` if unimportant

Example

```
SPLIT_ISSUE += "bidMFB:no:yes:fu(*/*/fuBRP,*/*/fuNOP)";
```

UARCH

The UARCH knob defines the microarchitecture generation. The only valid value is `merced`.

Index

H

I

K

L

M

O

P

R

S

T

U

V